Dynamische Modellanalyse von Metamodellen mit Operationaler Semantik

Dissertation

zur Erlangung des akademischen Grades
doctor rerum naturalium

im Fach
Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von
Herrn Dipl-Inf. Michael Soden

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II
Prof. Dr. Elmar Kulke

Gutachter/Gutachterin
1. Prof. Dr. Joachim Fischer
2. Prof. Dr. Holger Schlingloff
3. Prof. Dr. Andreas Prinz

Tag der Verteidigung: 12.05.2014

Danksagung

An dieser Stelle möchte ich einigen Personen besonders danken, ohne deren Hilfe und Unterstützung die vorliegende Arbeit nicht gelungen wäre.

Allen voran möchte ich meinem Doktorvater Prof. Dr. Joachim Fischer für die stete Betreuung, Motivation und Freiheit bei der Themenwahl danken. Durch seine wissenschaftliche Erfahrung hat er mir die entscheidenden Leitplanken bei der Bearbeitung gesetzt und mich auf vorhandene Unschärfen und Probleme aufmerksam gemacht, die man als Doktorand — mit der sich über die Zeit einstellenden Innenperspektive — nicht mehr wahrnimmt.

Fachlich war mir die Habilitationsarbeit von Prof. Dr. Andreas Prinz zu *Abstract State Machines* eine Orientierung, durch die ich einen besseren Zugang zu den Formalisierungen von Sprachsemantik erhalten habe. Ich bin besonders dankbar, dass er als Gutachter für diese Arbeit gewonnen werden konnte.

Zu besonderem Dank bin ich meinen langjährigen Weggefährten Hajo Eichler verpflichtet, mit dem ich das *Projekt Dissertation* begonnen habe. Die gemeinsamen Diskussionen, Reisen und Konferenzen haben die Inhalte dieser Arbeit entscheidend geprägt. Ebenso geht mein Dank an Dr. Markus Scheidgen, mit dem wir beide intensive "Meta-Diskussionen" aber auch "Ergebnis-Instanzen" hatten.

Des weiteren gilt mein Dank den Kommilitonen und Professoren aus dem Graduiertenkolleg METRIK. Die regelmäßigen Workshops trugen maßgeblich zur Schärfung und Eingrenzung des Themas bei. Namentlich möchte ich aus dieser Runde besonders Dr. Daniel Sadilek, Guido Wachsmuth und Dr. Stephan Weissleder hervorheben, mit denen der fachliche Austausch angenehm produktiv war.

Ganz herzlich möchte ich mich bei meinen Arbeitskollegen bedanken: Joachim Hößler für die kritische Reflektion bei der Themenfindung und während der Anfangsphase der Promotion sowie meinen Vorgesetzten Dr. Olaf Kath und Dr. Marc Born, die mir die notwendige Flexibilität für diese Arbeit eingeräumt haben.

Neben den Kollegen aus der Wissenschaft soll mein privates Umfeld nicht unerwähnt bleiben, dem ich mein Dankbarkeitsgefühl gegenüber zum Ausdruck bringen möchte. Dazu zählt mein Freund Bastian Claaßen, der mir in der kritischen Phase den nötigen Schub gegeben hat und nicht zuletzt meine Frau Iliane und meine Familie, die mich unterstützt haben und mir die nötige Kraft und Zeit gaben, diese Arbeit abzuschließen.

Zusammenfassung

Metamodellierung im Sinne der Meta Object Facility (MOF) stellt eine Methode für die strukturelle Definition der abstrakten Syntax von Modellierungssprachen und Modellen im Softwareentwicklungsprozess dar. Um Modellsimulation und dynamische Analysen für metamodellbasierte Sprachen zu unterstützen, fehlt es an einem Kalkül zur operationalen Semantik. In dieser Arbeit wird ausgehend von MOF die Aktionssemantik MActions entwickelt, die die Definition von operationaler Semantik als Verhalten in Metamodellen ermöglicht. Diese Erweiterung geht einher mit der Beschreibung von Laufzeitmodellen sowie Zuständen und Parallelitätseigenschaften, so dass eine Verifikation von dynamischen Eigenschaften möglich wird. Zu diesem Zweck wird mit der *Linear Temporal Object Constraint Language* (LT-OCL) exemplarisch eine prädikatenlogische Temporallogik entwickelt, die eine metamodellunabhängige Analyse für ausführbare Modelle erlaubt. Dabei ist die Semantik von temporalen Ausdrücken über Zuständsänderungen von (aufgezeichneten) Ausführungsläufen beschrieben, wobei eine Linearisierung parallele Änderungen zusammenführt. Als weiteren Anwendungsfall der dynamischen Analyse untersuchen wir die Relation zum Verhaltensvergleich im Sinne der Bisimulationstheorie. Metamodelle, Aktionssemantik und Temporallogik werden mittels einer erweiterten Abstract State Machine (ASM) formal beschrieben und kommen in zwei Fallstudien zur Anwendung (Timed Automata und C#).

Abstract

Object-oriented metamodelling as defined by the Meta Object Facility (MOF) provide a means to describe the structure of models and the abstract syntax of modelling languages at various stages in a software development process. However, MOF lacks concepts for the definition of operational semantics and there is no support for dynamic model analysis based on the semantics and abstract states of a language definition. This thesis investigates on extending the metamodelling framework with an action semantics — the MActions — to support the definition of operational semantics in metamodels and enable simulation as well as verification of dynamic properties. For this purpose, runtime models are incorporated with semantics for states, time, and properties of parallelism that allow a generic analysis solely bound to a certain metamodel definition. Furthermore, we develop the *Linear Temporal Object Constraint Language* (LT-OCL) to perform a dynamic analysis of execution runs based on the executable models. The semantics of this temporal predicate logic is bound to state changes of (recorded) execution traces that are linearizations of parallel changes of the runtimes model. This establishes the link to the theory of bisimulation as a second application case of dynamic analysis. Abstract State Machines (ASM) have been used to formally define the action language in conjunction with metamodels and the temporal logic. As proof of concept of the whole approach, the framework has been implemented and applied to two languages as case studies (namely Timed Automata and C#).

Inhaltsverzeichnis

Abkürzungsverzeichnis

ASM	Abstract State Machine
ASR	Abstrakter Syntaxraum
AST	Abstract Syntax Tree
CCS	Calculus of Communicating Systems
CLR	Common Language Runtime
CSP	Communicating Sequential Processes
CTL	Computation Tree Logic
DSL	Domain Specific Language
EA	Evolving Algebra
EMF	Eclipse Modeling Framework
EMP	Eclipse Modeling Project
GMF	Graphical Modeling Framework
GPSS	General Purpose Simulation System
GSOS	Grand Structural Operational Semantic
LTL	Linear Time Temporal Logic
LT-OCL	Linear Temporal Object Constraint Language
LTS	Labeled Transition System
LTTS	Labeled Terminal Transition System
LZR	Laufzeitraum
MDA	Model Driven Architecture
MOF	Meta Object Facility
MXF	Model Execution Framework
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query View Transformation
RTE	Runtime Environment
SGT	State Generating Transition
SOS	Strukturelle Operationale Semantik
SDL	Specification and Description Language
TLA	Temporal Logic of Actions
TGG	Triple Graph Grammar
TM	Turingmaschine
TTS	Terminal Transition System
UML	Unified Modeling Language
UTM	Universelle Transformationsmaschine
XMI	XML Metadata Interchange (Format)
XML	Extensible Markup Language

KAPITEL 1

Einleitung

Modellbildung und Simulation sind seit Jahrzehnten etablierte Techniken der Software- und Systementwicklung. Die Idee der modellgetriebenen Softwareentwicklung, wie z.B. die *Model Driven Architecture*-Initiative (MDA) der Object Management Group (OMG), geht einen Schritt weiter und rückt das Modell in den Mittelpunkt des gesamten Entwicklungsprozesses. Modellgetrieben (*engl. model-driven*) bezeichnet deshalb nicht nur die vollständige Generierung eines lauffähigen Programms aus Modellen, sondern insbesondere eine durchgängige Modellbildung in nahezu allen Prozessphasen, Automatisierung durch Modelltransformationen, Wiederverwendung von Modellen, integrierte Werkzeuglandschaften und Nachverfolgbarkeit (*engl. Traceability*) [1][2].

Bevor eine genauere Definition des Modellbegriffs für diesen Kontext gegeben wird (vgl. 1.1), soll der besondere Stellenwert von Sprachen zur Modellbeschreibung hervorgehoben werden. Das Spektrum der verwendeten Sprachen in der Softwareentwicklung reicht von natürlicher Sprache (z.B. Wortmodelle oder informelle Dokumente mit skizzenhafter Darstellung), über (semi-)formale Modellierungssprachen (z.B. Unified Modelling Language, Petri-Netze) bis hin zu Programmiersprachen [3]. Darüber hinaus kommen für verschiedene Fachgebiete heutzutage vermehrt sog. *Domain Specific Languages* (DSLs) zum Einsatz, die spezielle Konzepte zur Beschreibung von domänenspezifischen Phänomenen bereithalten [4]. Auch wenn historisch die einzelnen Spracharten ihre Wurzeln in unterschiedlichen Forschungsgebieten der Informatik haben, stellt die *Metamodellierung* eine universell einsetzbare Methode dar, um Sprachdefinitionen und verschiedenste Artefakte in Softwareentwicklungsprojekten präzise zu beschreiben [5][6][7]. Im Zentrum stehen Modelle als strukturierte Daten, die als Instanz eines Metamodells in Struktur und Verhalten charakterisiert sind. Die *Meta Object Facility* (MOF, [8]) der OMG zur objektorientierten Definition von Metamodellen hat sich dafür als Standard durchgesetzt [9][10][11][12]. Diese Konvergenz der Beschreibungsmittel für Modelle und Sprachen durch ein gemeinsames Metamodellierungs-Framework schafft die Grundlage zur einheitlichen Definition und Analyse von Modellen als „Sprachinstanzen“.

Abgesehen von der strukturellen Beschreibung durch Metamodelle besteht im Hinblick auf die *dynamische Semantik* (auch: Ausführungssemantik, vgl. 1.1.2) von Modellen (noch) kein einheitliches Vorgehen. An dieser Stelle setzt die vorliegende Arbeit an. Durch Erweiterungen von MOF um eine Operationale Semantik, den

MOF Actions (kurz: MActions), wird die Basis zur Modellausführung gelegt. Das daraus resultierende, erweiterte MOF-Framework stellt die Basis dieser Arbeit dar. Darauf aufbauend lassen sich verschiedene Methoden zur Analyse entwickeln, die an die operationale Semantik von Metamodellen gebunden sind und somit universell für verschiedenste Sprachdefinitionen einsetzbar werden. Im Fokus unserer Analyse sollen dabei Softwaremodelle und Softwaremodellierungssprachen stehen. Als weiteren Schwerpunkt entwickeln wir zur dynamischen Analyse eine Temporallogik, die *Linear Temporal Object Constraint Language*. Diese stellt exemplarisch eine Analysesprache zur automatischen Überprüfung von temporalen Ausdrücken über Ausführungsläufe dar.

Damit ist der Rahmen dieser Arbeit vorgezeichnet. Im Folgenden werden einleitend verwendete Begriffe eingeführt und eingeordnet, die eine präzise Kommunikation über den Forschungsgegenstand ermöglichen sollen. Da diese Arbeit sich zwischen den verschiedenen Forschungsgebieten Softwaretechnik, Simulation- und Systemanalyse sowie theoretischer Informatik einordnet, sei angemerkt, dass eine einheitliche Terminologie notwendigerweise mit Kompromissen einhergeht. Dieses Vorgehen erlaubt jedoch erst eine Einordnung in die unterschiedlichen Kontexte. Nach dieser Einleitung wird in Abschnitt 1.3 die Arbeitshypothese dieser Dissertation und der Lösungsansatz dargelegt. Der weitere Aufbau der Arbeit ist in Abschnitt 1.4 ersichtlich.

1.1 Modelle und Sprachen

Im Kontext der MDA sind *Modelle* Ausdruck der Funktion, der Struktur und/oder des Verhaltens eines Systems [1]. Modelle dienen zur Kommunikation über ein System[1] und sind synonym zum Begriff der *Spezifikation*. Modelle werden notiert und kommuniziert unter Zuhilfenahme von Sprachen (*Modellierungs-* oder *Spezifikations*sprachen). Laut MDA wird ein Modell als *formal* bezeichnet, wenn es über ein MOF-Metamodell definiert ist. Formal sollte hier nicht verwechselt werden mit dem Begriff eines formalen Systems, dem ein rigoroses mathematisches Modell zu Grunde liegt. MOF-Modelle (Instanzen von Metamodellen) stellen, wie in anderen Kontexten üblich, ein Hilfsmittel zum Umgang mit der Realität dar, d.h. sie bilden ein Original(-system) ab, unterliegen Techniken der Abstraktion und dienen einem Modellzweck (vgl. [13]). Dennoch besteht ein Modell nicht als Satz von Differentialfunktionen oder Gleichungssystemen, sondern als Objektgraph. Die Struktur dieses Objektgraphen definiert ein Metamodell mittels objektorientierter Konzepte, die im MOF-Standard vorgegeben sind, d.h. Klassen, Attribute, Assoziationen, usw. (vgl. [8]). Man spricht davon, dass ein Modell zu einem Metamodell *konform* ist, wenn es eine gültige Instanz im Sinne der MOF-Spezifikation ist. Die Instanziierung von MOF, und dabei insbesondere die Problematik der Metaebenen, werden uns eingehender in Kapitel 2.2 beschäftigen.

1.1.1 Metamodelle und Sprachdefinitionen

Um den Zusammenhang von Metamodellen und Sprachdefinitionen herauszustellen, wird eine genauere Definition von Sprachen, als sie in der MDA gegeben wird, benötigt. Die Elemente einer Sprache werden gängigerweise in Syntax, statische Se-

[1] System ist hier im Sinne einer Systemtheorie zu verstehen und meint nicht ausschließlich Software. Die MDA-Spezifikation sagt an dieser Stelle nichts genaueres aus

mantik und dynamische Semantik eingeteilt[2]. Auf Syntaxebene unterscheidet man die *konkrete* Syntax von der *abstrakten* Syntax (*engl. abstract syntax*), historisch begründet in der Verarbeitung von textuellen Sprachen durch Parser. Dabei produziert ein Parser als Resultat einen sog. *abstrakten Syntaxbaum* (*engl. abstract syntax tree*, AST), der akzeptierte Wörter als logische Baumstruktur repräsentiert. Der AST dient als Grundlage zur Überprüfung der statischen Semantik (*semantische Analyse*) und der weiteren Verarbeitung z.B. in einem Compiler. Heutzutage wird eine klare Definition von ASTs immer schwieriger, da das Abstraktionsniveau dieser Repräsentationsform vom Verwendungszweck und eingesetzten Werkzeugen abhängt[3]. Meist wird stufenweise eine logische Struktur in eine andere überführt, es finden also mehrere Transformationen der Eingabe statt. Zum Beispiel wird ein vom Lexer erzeugter Tokenstream in mehreren Durchgängen zum Syntaxbaum transformiert. Währenddessen finden u.U. Bezeichnerauflösungen statt oder syntaktische oder semantische Prädikate werden ausgewertet.

Metamodelle ermöglichen eines Definition dieser Strukturen, weshalb im Weiteren der Begriff der abstrakten Syntax einer Sprache gleichbedeutend verwendet wird zu dem Begriff des Metamodells. Dabei lassen sich z.B. die genannten Strukturen der verschiedenen Abstraktionsstufen beim Parsen entweder durchweg mit Metamodellen beschreiben, oder aber es steht am Ende dieses Prozesses ein gültiges Modell, das einer Metamodelldefinition genügt. Diese Sicht lässt sich auch auf graphische Sprachen übertragen, die keine Ausnahme bilden[4]. Wir gebrauchen im Folgenden den neutraleren Begriff der *Notation* oder *Repräsentation* synonym zu dem der konkreten Syntax (anstatt „konkrete Syntax“ verwenden wir ferner einfach „Syntax“ als Kurzform). Es wird angenommen, dass es zu den Konzepten, die in einem Metamodell definiert sind, eine oder mehrere Repräsentationsformen gibt, die dem jeweiligen Anwendungskontext angepasst sind. So wird z.B. für Metamodelle dem MOF-Standard entsprechend eine Teilsprache von UML als Notation verwendet. Es wird ferner angenommen, dass ein Metamodell – dem Anwendungszweck entsprechend – für jede Sprache angegeben werden kann. Bespiele für solche Metamodelldefinitionen existieren bereits für eine Vielzahl von Sprachen (u.a. UML [3], OCL [17], QVT [18], SDL [19], Java, C/C++, C# [20]).

Mit Blick auf die statische Semantik ergeben sich als Konsequenz aus dieser Definition unmittelbar die folgenden Eigenschaften:

- Die statische Semantik spiegelt sich zu einem gewissen Teil im Metamodell wider, da ein Metamodell i.Allg. nicht nur einen einfachen Syntaxbaum definiert, sondern einen getypten Graphen. Im Umkehrschluss muss eine gewisse semantische Analyse bereits *vor* der Erzeugung einer gültigen Metamodellinstanz stattfinden.
- Die Regeln der statischen Semantik einer Sprache lassen sich unter zu Hilfenahme der Object Constraint Language (OCL) spezifizieren und überprüfen. Dies gilt für alle Regeln, die sich als Bedingungen in Prädikatenlogik erster Stufe ausdrücken lassen.
- Die syntaktische und semantische Analyse einer Sprache bleibt zwar weiterhin bestehen, der Fokus für eine Sprachanalyse (und insbesondere der dynamischen Semantik) richtet sich jedoch nur noch auf das Modell. Syntax und

[2] die Pragmatik spielt für die technischen Betrachtungen in der Informatik i.Allg. eine untergeordnete Rolle, vgl. [14]

[3] so lässt z.B. ANTLR eine Annotation der Struktur sowie Aktionen zu deren Gestaltung an die Grammatik zu. Somit lassen sich u.U. direkt Graphstrukturen beim Parsen erzeugen, vgl. [15]

[4] Auch wenn formale Definitionen graphischer Sprachen selten sind, existieren vergleichbare Verfahren zu deren Beschreibung mittels Graphgrammatiken, z.B. GenGed [16]

Sprachkonstrukte sind entkoppelt und die statische Semantik ist letztendlich Teil desselben Modells.

Dieses Verständnis der abstrakten Syntax betont den Übergang von der Grammatik- und syntaxbezogenen Sicht auf eine Sprache hin zu einer konzeptbezogenen Definition der Sprachstruktur: anstatt über eine Formale Grammatik die Syntax der Sprache zu definieren, liegt der Fokus auf der Struktur der abstrakten Syntax. Zudem ist ein Rahmenwerk gegeben, in dem sich sowohl textuelle Sprachen als auch graphische Sprachen mit denselben Mitteln beschreiben lassen.

1.1.2 Dynamische Semantik

Neben der strukturellen Beschreibung der abstrakten Syntax und statischen Semantik definiert die dynamische Semantik das Verhalten einer Sprache. In der Informatik spricht man verkürzt meist einfach von *Sprachsemantik* (auch *Ausführungssemantik*), weil der Anwendungskontext fast ausschließlich die Ausführung eines Programms ist. Hierzu wurden im Laufe der Jahrzehnte eine Vielzahl unterschiedlicher Ansätze entwickelt, die sich in drei Kategorien einteilen lassen:

Axiomatische Semantik Abgeleitet aus der mathematischen Logik und Modelltheorie bilden Axiome ein formales System zur Sprachbeschreibung. Eine axiomatische Definition durch Schlussregeln erlaubt eine Herleitung (d.h. Beweisen) von Eigenschaften mittels eines Theorembeweisers. Es lassen sich nicht nur Korrektheitsbeweise von Programmen erstellen (z.B. wie im Hoare-Kalkül), sondern auch die Korrektheit von Spracheigenschaften nachweisen (z.B. Typsicherheit).

Denotationelle Semantik Eine denotationelle Definition der Sprachsemantik ist eine Abbildung der Sprachkonstrukte auf ein mathematisches Objekt oder auf eine Sprache mit formaler Semantik. Somit wird die Bedeutung der Sprache mittels einer *Interpretationsfunktion* zurückgeführt auf die Bedeutung und Eigenschaften eines bekannten Objekts (meist Funktionen, deshalb auch *Funktionssemantik*). Letzteres wird auch als semantische Domäne (*engl. semantic domain*) bezeichnet.

Operationale Semantik Das Verhalten einer Sprache wird als Interpretation auf einer abstrakten Maschine (Turing Maschine o.Ä.) definiert, meist angegeben als schrittweise Zustandsänderung der abstrakten Maschine. Als Spezialfall gilt die von Plotkin entwickelte *Strukturelle Operationale Semantik* (SOS), die durch Transitionsregeln (syntaxorientierte Schlussregeln) Zustandsübergänge beschreibt [21].

Es sei angemerkt, dass sich diese drei Methoden naturgemäß überschneiden. So ist z.B. in der denotationellen Semantik eine Axiomatisierung des mathematischen Objekts gegeben und die SOS bildet ein formales System von Schlussregeln. Die Operationale Semantik trägt darüber hinaus Züge eines Übersetzungsprozesses von der (abstrakten) Syntax einer Sprache in ein formales System. Allen dreien ist eine rigorose mathematische Formalisierung gemein.

Mit Blick auf die Praxis lässt sich eindeutig eine Tendenz zur Anwendung der operationalen Semantik feststellen. Dies hängt vermutlich damit zusammen, dass die exakte Beschreibung von Ausführungsaspekten insb. für die Vergleichbarkeit von Implementierungen einer Sprache zunächst im Vordergrund steht. Angefangen mit der ALGOL 60-Spezifikation [22] wurden verschiedenste Programmier- und Spezifikationssprachen (teilweise) formal definiert, so zum Beispiel C++ [23], Java [24],

C# [25], SDL [26], UML [27][28][29]. Für die Formalisierung gibt es im Allgemeinen kein einheitliches Vorgehen. Die wohl am weitesten verbreitete Methode ist die der *Abstract State Machines* (ASMs [30][31]). Das von Gurevich et al. entwickelte mathematische Modell basiert auf algebraischen Strukturen, die abstrakte Zustände repräsentieren und deren Trägermengen und Operationen sich über die Ausführungszeit ändern. Die *Abstract State Machine Language* stellt eine Umsetzung in Richtung einer formalen Spezifikationssprache dar (AsmL [32]). Daneben existieren vergleichbare Herangehensweisen die sich auf attributierte Graphen und Graphtransformationen stützen (z.B. [29]).

Neben formalen Methoden wird oftmals ausschließlich natürliche Sprache zur Beschreibung der dynamischen Semantik verwendet (z.B. die Programmiersprachen C [33] oder C# [34]). So definiert z.B. die UML2-Spezifikation jegliches dynamische Verhalten rein in englischer Sprache, was nicht zuletzt bei den über 40 *Semantic Variation Points* zu mehrdeutigen Interpretation führt [35][3]. Es mag überraschen, aber abgesehen von ALGOL 60 wurde eine formale Definition einer Sprache fast immer *nach* deren Implementierung und Verbreitung erstellt[5]. Ein Grund hierfür ist die Tatsache, dass eine formale, semantische Spezifikation nicht unbedingt eine geeignete Vorlage für Implementierungen bietet. In der Praxis wird deshalb eine „pragmatische“ Spezifikation einer formalen vorgezogen und u.U. Aspekte einer Sprache nachträglich formalisiert, um Korrektheitsbeweise zu führen.

Heutzutage dient der Einsatz von Methoden zur operationalen Semantik neben der präzisen Definition der dynamischen Semantik vielfach ausschließlich der unmittelbaren Ausführbarkeit. Der Trend weg von Allzwecksprachen hin zu immer neuen domänenspezifischen und angepassten Modellierungssprachen schafft eine Notwendigkeit für pragmatische Ansätze, die – wenn auch nicht immer mathematisch formal fundiert – eine schnelle Werkzeugentwicklung unterstützen. In diese Kategorie fällt auch die von Watt und Mosses entwickelte Methode der *Action Semantics* (*dt. Aktionssemantik*, vgl. [37]), die den Gesichtspunkt der Programmausführung betont.

1.1.3 Operationale Semantik für Metamodelle

Wesentliche Probleme der hier betrachteten metamodellbasierten Sprachdefinition sind eine fehlende mathematisch formale Definition von MOF sowie die konzeptionelle Trennung zwischen Notation und abstrakter Syntax. Die traditionell syntaxorientierten Ansätze, entwickelt für textuelle Sprachen, lassen sich nur mit Mühe auf Metamodelle übertragen, die eine standardisierte, graphische UML-Notation besitzen. Auch wenn eine Einbettung graphischer Elemente einer Sprache in eine formale Beschreibungsform denkbar ist (wie z.B. von Börger für eine Definition der UML-Aktivitäten mittels ASM gezeigt, vgl. [27]), bezieht sich die Beschreibung auf die Notation und nur indirekt auf das Metamodell. Zudem setzt die fehlende Formalisierung der graphischen Syntax eine gewisse Intuition für die Einhaltung der Gültigkeit von Diagrammen voraus. Eine andere Lösung besteht in der Überführung des Metamodells in ein formales System, wie es z.B. der OCL-Standard macht (vgl. Appendix A in [17]), wobei denotationell alle Konzepte auf mathematische Objekte abgebildet werden; vorstellbar wäre auch eine vergleichbare Vorgehensweise für die Beschreibung der operationalen Semantik.

Eine Reihe von Ansätzen beschäftigt sich mit dieser Problematik. Sie bieten neben (semi-)formalen Methoden durch Übersetzung in bekannte Kalküle auch direkte Annotationen von Semantikbeschreibungen in Metamodellen als Lösung. Mit

[5] vgl. [36]

operationaler Semantik in Metamodellen sind Ansätze gemeint, die Techniken zur Definition innerhalb der Metamodellierungsarchitektur MOF bereitstellen, ohne in eine semantische Domäne abzubilden. Diese Ansätze verfolgen ähnliche Formen von Aktionssprachen, Graph- oder Modelltransformationen wie die in dieser Arbeit eingeführten MActions (z.B. [38],[39],[29],[40]). Diese Ansätze, welche teilweise als *dynamische Metamodellierung* bezeichnet werden (Engels et al.), sind im Allgemeinen dadurch charakterisiert, dass sie operationale Semantik als (Spezialfall der) Modelltransformation betrachten und Konzepte zur schrittweisen Modelländerung einführen.[6]

Aus dieser Motivation heraus und dadurch dass durch Erweiterung der Metamodellierung um eine operationale Semantik eine Reformalisierung unnötig und Modelle direkt ausgeführt und weitergehend (dynamisch) analysiert werden können, entwickeln wir in Kapitel 2 und 3 die MActions als Aktionssemantik. Eine ausführlichere Diskussion der Grundproblematik der Anwendung der etablierten Kalküle SOS und ASM sowie verwandter Arbeiten folgt in Kapitel 5.

1.1.4 Modellausführung, Laufzeitmodell

Aus diesen Überlegungen heraus bezeichnen wir im Folgenden mit dem Begriff der *Modellausführung* (*engl. model execution*) allgemein den Prozess der schrittweisen Abarbeitung eines Modells durch eine (abstrakte) Maschine. Dabei entspricht dieser Prozess der operationalen Semantik des Modells und deckt sich mit der nicht weiter spezifizierten Begrifflichkeit der *model execution* im Kontext der MDA (vgl. [1][41][42]). Die Sprache zur Definition der Modellveränderung ist dabei zunächst zweitrangig.

Wie bei der Programmausführung äußert sich das Verhalten eines Modells bei seiner Ausführung durch Änderungen der Konfigurationen. Wir bezeichnen deshalb mit *Laufzeitmodell* (*engl. runtime model*) die Objekte eines Modells, die sich bei der Modellausführung über die Zeit ändern (Zustandsgrößen). Dadurch grenzen wir das Laufzeitmodell von der abstrakten Syntax ab, indem wir alle Elemente zur abstrakten Syntax zählen, die sich während der Modellausführung nicht verändern und somit statisch sind. Diese Einteilung entspricht der Metapher vom Programm und Prozess, bei der der Prozess alle zum Ablauf benötigten Verwaltungsinformationen mit sich führt. Konsequenterweise behalten wir den Begriff der *Ausführungsumgebung* (*engl. execution environment*, vgl. [1]) auch für Modelle bei.

Weitergehende Betrachtungen zur Modellausführung sowie der Definition des Laufzeitmodells finden sich in Kapitel 3.

1.1.5 Simulation vs. Modellausführung

Da die Modellausführung durch operationale Semantik zu Überschneidungen mit dem Begriff der Simulation führt, soll in diesem Abschnitt kurz eine genauere Einordnung und Abgrenzung der bisher vorgestellten Konzepte gegenüber der Simulation gegeben werden. Vorweg sei angemerkt, dass der Begriff Simulation ein in sich heterogenes Forschungsfeld bezeichnet und hier nur die allgemeine Terminologie geklärt werden kann [43].

[6] Einige Autoren benutzen auch den Begriff der *ausführbaren Metamodelle* (*engl. executable metamodels*). Diese Begrifflichkeiten sind natürlich wörtlich genommen unzutreffend, da die Metamodelle selber nicht ausgeführt werden, sondern die Modelle, die sie definieren – sie betonen vielmehr die Ausführungssemantik innerhalb der Metamodellierung

Definiert man Simulation vereinfacht als experimentelles Analyseverfahren mit den Phasen Modellbildung, Modellausführung und Auswertung (vgl. [44]), so ist die modellgetriebene Softwareentwicklung ein Spezialfall der Simulation im Hinblick auf die Modellausführung. In diesem Fall ist das untersuchte System *Software*. Gemeinsamkeiten können bei allen Phasen gefunden werden, wobei der *Modellzweck* eines Softwaremodells als Abstraktion einer zu entwickelnden Software fast ausschließlich der Spezifikation dient. Somit geschieht die Modellbildung *im Vorfeld als Referenz* einer möglichen Implementierung, wohingegen Simulationsanalysen durch Experimente eines gegebenen Softwaresystems eher selten sind.

Softwaresimulation existiert meist dort, wo Anforderungen durch ein exaktes Modell präzisiert werden, eine vollständig lauffähige Spezifikation kostengünstiger als eine direkte Entwicklung ist, oder aber der Entwicklung auf der Zielplattform externe Faktoren im Wege stehen (z.B. Verfügbarkeit der Hardware bei der Steuergeräteentwicklung in der Automobilindustrie, o.Ä.). Dies trifft insbesondere auf den Bereich der eingebetteten (missionskritischen) Systeme zu, wo sich Softwaresimulation mittels spezialisierter Werkzeuge und eigenen Modellbeschreibungssprachen (Simulationssprachen) etabliert hat, um zudem Analysen des Modells im Vorfeld zu ermöglichen (z.B. MATLAB Simulink, SCADE oder PragmaDev SDL-RT [45][46][47]).

Bei der Simulation eingesetzte Simulationssprachen sind an den Anwendungskontext angepasste, spezialisierte Programmiersprachen und können deshalb als DSLs betrachtet werden (vgl. [48][49][50]). Historisch gesehen haben sie sich aufgrund ihrer auf die Simulation zugeschnittenen Konzepte herausgebildet (z.B. GPSS oder Simula [51]). Charakteristische Eigenschaften von Simulationssprachen sind u.a. die Handhabung von Zeit als unabhängige Modellgröße (kontinuierlich vs. diskret), Zufälligkeit von Ereignissen, der Umgang mit Nebenläufigkeit (Prozessinteraktionen, Ereignisbehandlung, parallele Simulation) und die Behandlung von Ein- und Ausgabedaten (vgl. [52]). Zudem kommt der Visualisierung des Systemzustandes durch *Animationen* eine besondere Rolle zu. Umgekehrt hatte die Entwicklung von Simulationssprachen einen direkten Einfluss auf die Evolution der Programmiersprachen, so haben z.B. Smalltalk und später C++ die Abstraktionsmechanismen der Objektorientierung aus Simula übernommen (vgl. [53]). Neben der Vielzahl von spezialisierten Simulationssprachen und -werkzeugen werden oftmals Programmiersprachen in Kombination mit (Simulations-)Bibliotheken eingesetzt (z.B. [54]). Diese bieten verschiedene Abstraktionen zur effizienten Formulierung eines Simulationsproblems oder der Visualisierung an [55]. Da die dynamische Semantik letztlich durch die verwendete Programmiersprache vorgegeben ist, bedarf es hier keiner zusätzlichen Konzepte zur Definition der Ausführungssemantik.[7]

Daraus gefolgert bilden Simulationssprachen in Bezug auf die dynamische Semantik keine Ausnahme und die in Abschnitt 1.1.2 eingeführten Konzepte lassen sich unmittelbar übertragen. Wie bei Programmiersprachen wird die operationale Semantik in gleicher Weise definiert, auch wenn der (Modell-)Zeit ein besonderer Stellenwert zukommt. Die Metamodellierung zur Definition der abstrakten Syntax ergänzt um operationale Semantik kann somit ebenfalls als Basis für Simulationssprachen dienen.

In Bezug auf die Terminologie basiert in dieser Arbeit Simulation immer auf der in Abschnitt 1.1.4 und 1.1.3 beschriebenen Form der Modellausführung. Demnach ist ein *Simulator* ein Werkzeug zur Modellausführung im Rahmen einer Metamodellierungsarchitektur, ein *Simulationslauf* (syn. *Modelllauf*, *Simulationsexperiment*) das

[7] Die Realisierung mittels einer Allzweck-Programmiersprache kann auch als sog. *interne DSL* betrachtet werden (vgl.[4])

ablaufende Verhalten über einen bestimmten Zeitraum im Simulator (vgl. [56]) etc.

1.1.6 Produkt vs. Modell

Bei der Softwaresimulation erscheint die klassische Trennung von *Produkt*, als Synonym für das zu entwickelnde System und *Modell* aufgehoben, zumal beide virtuell im Speicher eines Rechners existieren, durch eine Programmier- und Modellierungssprache ausgedrückt sind und Verhalten beschreiben. Dennoch unterliegen Modelle Techniken der Abstraktion, d.h. sie beschreiben nur die Aspekte des Systems, die für den jeweiligen Zweck gefordert sind. Dadurch können im Allgemeinen Aussagen über Eigenschaften eines Modells nicht ohne weitere Prüfung auf das zu entwickelnde Produkt übertragen werden.

Betrachtet man den Ausführungsaspekt, kann eine Softwaremodellausführung zunächst prinzipiell nicht von einer Produktausführung (Implementierung) unterschieden werden. In besonderen Fällen kann das *beobachtete Verhalten* aus Sicht eines externen Betrachters sogar zwischen Produkt und Modell identisch sein und sich lediglich durch genaueres betrachten der Zwischenschritte der ausführenden Maschine (oder der Performance) unterscheiden. Um eine klare Unterscheidung der Begrifflichkeiten zu erhalten, soll unter Produkt im weiteren ein *binäres Programm* verstanden werden, das im Allgemeinen auf einer Hardware ausgeführt wird[8] und das sich vom Softwaremodell durch seinen (Anwendungs-)Zweck unterscheidet (vgl. 1.1.5). Das Produkt ist demnach das Ergebnis eines Kompilierungsprozesses durch einen Compiler und automatisch abgeleitet aus der Implementierung (als Quelltext[9], Implementierungsmodell). Es stellt das letzte Glied in der Entwicklungskette dar. Das Produkt ist durch seinen Anwendungskontext im realen System charakterisiert, während das Modell in einer künstlichen Umgebung der Analyse dient und nach Bedarf von Implementierungsdetails abstrahiert.

Es sei angemerkt, dass die Art der Ausführung durch einen Interpreter oder Compiler keine Rolle für die vorangehende Definition spielt. Eine Ausführung durch einen Interpreter ist lediglich durch den fehlenden Übersetzungsprozess abzugrenzen, wobei bei einem Kompilierungsvorgang ein Programm oder Modell zunächst in eine andere Repräsentation überführt wird, welche anschließend unmittelbar auf einer Maschine ausgeführt wird[10].

Wir unterscheiden ferner den Kompilierungsvorgang als die operationale semantikerhaltende Spezialform eines (Code-) *Generators*. Letzterer produziert aus einem Modell entweder ein anderes Modell, Quelltext oder eine kodierte Form zur Ausführung und realisiert somit eine Modelltransformation. Insbesondere fordern wir von einem Generator – im Gegensatz zu einem Compiler – nicht, dass er die dynamische Semantik erhält. Dies wird insbesondere in Grenzbereichen von z.B. Echtzeitanforderungen deutlich, wenn simulierte Kommunikation durch realen Austausch von Daten im System von Bandbreite, Durchsatz oder Prozessortaktung abhängt.

[8] Als Sonderfall rechnen wir auch *Bytecode*-Repräsentationen eines Programms dazu, wie z.B. bei der *Java Virtual Machine* oder der *Common Language Runtime* der .NET Plattform (vgl. [57][58]).

[9] Quelltext wird als Sonderfall eines Modells aufgefasst, vgl. 2.1

[10] Diese Trennung kann natürlich die Vielfalt der interpretierten Sprachen und Hybridausführungsumgebungen wie das Just-In-Time kompilieren etc. nicht in ihrer Gänze erfassen. Streng genommen gibt es *keinen* Unterschied zwischen Modellausführung und Programmausführung – die Terminologie ist lediglich historisch zu begründen durch die Interpretation eines Programms durch Hardware bzw. durch Software

1.1.7 Dynamische Analyse

Dynamische (Programm-)Analyse (*engl. Dynamic Analysis, DA*) bezeichnet die Analyse von Eigenschaften eines ausführenden Programms [59]. Im Gegensatz zur statischen Analyse, die ohne eine Ausführung auskommt und in der Regel alle potentiell möglichen Ausführungsläufe eines Programms berücksichtigt, dient ein oder einige wenige Abläufe als Grundlage der Analyse. Demnach gleicht die Aussagemächtigkeit einer DA dem von Testläufen. Deshalb soll zunächst kurz der Begriff des Testens beleuchtet und gegenüber der DA und der Simulation abgegrenzt werden.

Testen ist ein Vorgehen, bei dem ein System oder eine Komponente (1) unter vorbestimmten Bedingungen ausgeführt wird, wobei (2) Beobachtungen oder Aufzeichnungen vorgenommen werden, um (3) diese anschließend auszuwerten (vgl. [60]). Vergleicht man diese allgemeine Definition mit der der Simulation (s. Abschnitt 1.1.5), so ergeben sich methodische Überschneidungen in den Punkten (1) bis (3). Tatsächlich unterscheidet sich der methodische Ablauf einer Simulation vom Testen nur durch ihren experimentellen Charakter, bei dem der Ausgang i.Allg. offen ist. Beim Testen wird ein bestimmtes Ergebnis erwartet, um ein Produkt- oder Modellverhalten zu bestätigen oder zu falsifizieren, was dann als Erfolg oder Misserfolg eines Tests gewertet wird. Simulationsexperimente hingegen suchen (ergebnisoffen) nach Gleichgewichtszuständen, (In-)Stabilitäten, Empfindlichkeiten der Systemparameter des Modells oder gegenüber Eingaben vor dem Hintergrund von Langzeiteffekten etc. (s.a. [13]).

Der Softwaretest im Besonderen validiert eine Implementierung gegen ihre Spezifikation, d.h. das Erfolgskriterium der Tests ist durch die Spezifikation vorgegeben. Setzen wir die Spezifikation mit Modellen gleich (wie es in der MDA getan wird), so stellt der Softwaretest abstrakt den Vergleich eines Modellverhaltens gegen das tatsächliche Produktverhalten dar. Im Hinblick auf die Unterscheidung zwischen Modell und Produkt zielt Testen also auf das Produkt (oder Teile dessen) ab. Im Weiteren verstehen wir deshalb unter Testen den Vorgang der Validierung des Produkts, im Gegensatz zum *Modelltest*, der als Spezialfall das Modell zum Testgegenstand hat. Unter dem weitgedehnten Begriff des *Modellbasierten Testens* (*engl. model-based testing*)[11] fasst man alle Ansätze zusammen, die im Kern das Ableiten von Testfällen aus Modellen gemeinsam haben (vgl. [61]). Dies subsumiert z.B. das Generieren von Testfällen oder Testdaten direkt aus einem Spezifikationsmodell, das Beschreiben von Testspezifika durch Erweiterungen im (oder separat zu) einem Modell (z.B. als Annotationen oder Testmodelle) sowie Ansätze zur Interpretation von Spezifikationsmodellen während des Tests (z.B. *passives Testen*, *back-to-back*-Tests).

Dynamische Modellanalyse fußt – genau wie Testen – auf der Beobachtung, Messung und Auswertung von (abgeleiteten) Zustandsgrößen und deren Entwicklung über die Zeit. Basis ist also bei beiden die Modellausführung. Der Unterschied gegenüber dem Testen kann zum einen in der methodischen Zielsetzung sowie zum anderen in der Herangehensweise gesehen werden. Die DA ist in vielen Fällen ergebnisoffen, versucht qualitative oder quantitative Eigenschaften während der Modellausführung zu prüfen oder zu finden und zeigt somit eher starke Ähnlichkeiten mit dem Simulationsexperiment. Ferner zielt sie nicht (ausschließlich) auf das Produkt. Noch wichtiger ist die Unterscheidung in der Beschreibung von *erwartetem Verhalten*. Dazu werden formale, meist modale Logiken verwendet, um Aussagen über gültige Analyseszenarien oder Eigenschaften zu formulieren, die dem Modellchecking entlehnt sind. Beispiele umfassen u.a. Zustandsautomaten oder zeitliche

[11] als Synonym auch *modellgetriebenes* Testen (*engl. model-driven-testing*)

Prädikate/Pfadausdrücke, die gültiges Verhalten beschreiben. Es werden somit insb. keine Assertions in zu durchlaufende Szenarien wie in Testfällen eingefügt, sondern vielmehr indirekt über zeitgebundene Prädikate ausgedrückt. Unter dem Aspekt der Prüfung im Sinne eines *pass/fail*-Verhaltens lässt sich die DA als Methode zwischen Modellchecking und Testen einordnen, mit Überschneidungen zu letzterem, wenn formalisierte Testmodelle zum Einsatz kommen.

Als Technik werden bei der DA Zustandsgrößen in Form von *Programmpfaden* (engl. *Traces*) aufgezeichnet und ausgewertet. Diese Form der Messung von Programmzuständen ist zentraler Bestandteil der DA und ein weiterer Aspekt, der sie vom Testen mit Zusicherungen unterscheidet. In Abgrenzung zum Modellchecking wird also nicht ein kompletter Zustandsraum gezielt durchsucht, sondern es werden vielmehr einzelne Programmabläufe analysiert (es findet also insb. keinerlei *Backtracking* statt). Darüber hinaus existiert die *Runtime Verification (RV)* als Teilgebiet der DA, welche in den letzten Jahren zunehmend populärer wurde [62][63]. Ziel der RV ist es, das Vertrauen in eine korrekt funktionierende Implementierung durch kontinuierliche Überwachung (engl. *monitoring*) eines Programms zu steigern und bei Abweichungen ggf. sogar einzugreifen.[12] Dies umfasst sowohl Techniken zur *online* Überwachung während der Programmausführung, als auch *Offline*-Auswertung nach der Ausführung.

Für Metamodelle mit operationaler Semantik existieren hinsichtlich dynamischer Analysen nur wenige Forschungsarbeiten (s.a. 5.4). Um für Metamodelle dynamische Analysen durchführen zu können, bedarf es einer genaueren Betrachtung des Laufzeitmodells, der erzeugten Zustände sowie des Aufzeichnens von Zuständen die während Modellausführung entstehen. Dieser Problematik widmet sich Kapitel 4. Dort wird des Weiteren auch ein Ansatz zum Verhaltensvergleich erarbeitet, der über die DA hinaus zum Zwecke von Modelltests eingesetzt werden könnte.

1.2 Stand der Technik und Wissenschaft

Bevor wir zur Zielsetzung der Arbeit und Formulierung der Arbeitshypothese kommen, soll der zum Zeitpunkt dieser Dissertation in Forschung und Entwicklung erreichte Stand der Technik kurz zusammengefasst werden, um die in dieser Arbeit vorgestellten Neuerungen und Erkenntnisse besser bewerten zu können.

Während die Idee der modellgetriebenen Softwareentwicklung verschiedenste Bereiche der Informationstechnologie durchdrungen hat, wird ein Großteil heutiger Software (noch) nicht modellbasiert entwickelt [64]. Hierfür gibt es vielfältige Gründe. Moderne Programmiersprachen bieten Abstraktionsmechanismen, die mit denen von Modellierungssprachen vergleichbar sind. Zudem verfügen sie über mächtige Werkzeugunterstützung, Frameworks und Bibliotheken, welche oftmals noch für Modellierungssprachen fehlen. Eine Implementierung, welche auf einer informellen Spezifikation basiert, verspricht somit kürzere Entwicklungszeiten als eine vollständige Modellierung des Verhaltens der Software, obwohl Studien Gegenteiliges belegen [65][66]. Dahingegen hat die Metamodellierung bei der Entwicklung von Werkzeugen und Softwaresystemen Einzug erhalten, zumindest was die strukturierte Definition von Artefakten betrifft (siehe z.B.[5][12][67]). Die Beschreibung, Verwaltung und Pflege von Modellen, welche auf Metadatendefinitionen fußt, stellen bereits etablierte Techniken in der Entwicklung dar. Dabei kommen mehr und mehr spezialisierte Constraint- und Transformationssprachen zum Einsatz (z.B. [68][69][70]).

[12] Diese Techniken fallen letztlich in die gleiche Kategorie wie z.B. Sicherheitsmechanismen in sicherheitskritischen Systemen.

Im Zuge einer zunehmenden Zahl von Modellierungssprachen, Programmiersprachen, DSLs etc. kommt der Entwicklung von Werkzeugen und -umgebungen für diese Sprachen eine immer größere Bedeutung zu. Die Vielfalt sowie der Anspruch an heutige Sprachen reichen dabei von der Beschreibung einfacher, sequentieller Abläufe bis hin zu reaktiven und dynamischen Prozessen, die mittels althergebrachter Kalküle zur operationalen Semantik nur schwer fassbar sind. Bedenkt man den Einsatz der unterschiedlichen Sprachen in verschiedenen Phasen eines Entwicklungsprozesses, ergeben sich daraus (Sprach-)Integrationsprobleme. Neben diverser, integrierter Sichten auf ein oder mehrere Modelle in unterschiedlichen Notationen spielt die präzise, menschenlesbare und adaptierbare Definition der dynamischen Semantik eine Schlüsselrolle, um Ausführungsaspekte einer neuen Sprache eindeutig zu beschreiben und ggf. zu integrieren.

In der Praxis zeigt sich eine Diskrepanz zwischen der objektorientierten Beschreibung durch Metamodelle und den etablierten formalen, meist mathematisch-algebraischen Kalkülen zur Definition operationaler Semantik, da beide Techniken nicht zuletzt historisch separaten Forschungsfeldern entsprungen sind. Diese Kluft, manchmal auch auf die Formel *Modelware* vs. *Grammarware* gebracht, stellt für Sprachentwickler eine alltäglich neu zu überwindende Hürde dar und zwingt Entwickler zum Ausweichen und reformalisieren der selben Konzepte in anderen Notationen. Zudem werden verbreitete Kalküle wie ASMs oder formale Schlussregeln wie SOS oftmals als kryptisch empfunden und fügen sich in ihrer Herangehensweise nur unzureichend in die „Metamodellierungswelt“ ein. Die existierenden Kalküle und Techniken fokussieren darüber hinaus immer nur auf *eine* (Modellierungs-)Sprache und bieten keinerlei Möglichkeiten sprachunabhängig (d.h. metamodellunabhängig) Modellsimulationen in Form von dynamischen Analyse auszuwerten.

1.3 Ziele der Arbeit

Aus der dargestellten Problemlage heraus ergeben sich mehrere Ziele für die vorliegende Arbeit:

1. Bereitstellung geeigneter Modellierungstechniken für MOF-Metamodelle zur Definition operationaler Semantiken, so dass Ausführungsaspekte präzise beschrieben und Modelle simuliert werden können. Neben der abstrakten und konkreten Syntax vervollständigen diese Konzepte das MOF-Metamodellierungsframework und erlauben somit umfassende Sprachdefinitionen aller Aspekte einer Sprache, ohne auf andere Formalismen ausweichen zu müssen. Im einzelnen bedeutet dies:
   (a) Identifikation von fehlenden Konzepten zur Modellsimulation in bislang statischen MOF-Modellen.
   (b) Analyse und Vergleich existierender Techniken zur Metamodellierung. Dazu sollen insb. diese Fragen beantwortet und diskutiert werden:
      i. Können etablierte Kalküle wie z.B. ASMs genutzt bzw. für Metamodelle adaptiert werden? Wo ergeben sich Vor- und Nachteile?
      ii. Welche Konsequenzen ergeben sich aus der Metadatenhierarchie für operationale Semantiken und wie gliedern sich Modelle für Ausführungsaspekte in die Metaebenen ein?
      iii. Gibt es Möglichkeiten, die Akzeptanz durch Sprachdesigner mittels verbreiteter Notationen zu erhöhen (z.B. Standard-UML)?

   (c) Einordnung und Bewertung der neuen Techniken in Bezug auf herkömmliche Modellsimulation.
      i. Wie sehen Simulationen mit den neuen Techniken aus und können ausgewertet werden (s.a. 2.)?
      ii. Wie lassen sich typische Simulationseigenschaften wie Zeit, Parallelität, Zustände, diskrete und kontinuierliche Simulationen etc. abbilden?
      iii. Ergeben sich Komplexitätsprobleme durch Aspekte operationaler Semantiken in Metamodellen und sind diese überhaupt in einer Implementierung brauchbar?

2. Entwicklung einer metamodellunabhängigen Methode zur dynamischen Analyse von Modellen, um eine Validierung während und nach der Modellausführung zu ermöglichen. Dabei soll darauf verzichtet werden, Modelle in anderer Form zu reformalisieren und stattdessen eine ad-hoc Überprüfung dynamischer Eigenschaften ermöglicht werden:
   (a) Wie können dynamische Eigenschaften über beliebige operationale Semantiken und Metamodellen beschrieben werden?
   (b) Wie können Eigenschaften wie Zustände, Parallelität und Zeit generisch in der Auswertung berücksichtigt werden?
   (c) Wo zeigen sich Parallelen zu existierenden Techniken und Kalkülen, die sich nicht im Kontext einer Metadatenhierarchie bewegen? Wo ergeben sich Vor- und Nachteile?

3. Formalisierung der entwickelten Modellierungstechniken. Es soll hierbei um ein solides theoretisches Fundament für die dynamischen Analyse gehen, darüber hinaus soll auch ein Vergleich und eine Einordnung zu etablierten Kalkülen zur operationalen Semantik geliefert werden. Dabei sind folgende Fragen zu diskutieren:
   (a) Wie können statische und dynamische Aspekte von Metamodellen einheitlich mathematisch formalisiert werden?
   (b) Ergeben sich Vorteile durch eine Nutzung von Metamodellen gegenüber formalen Kalkülen?

Langfristig sollen die in 1. bis 3. untersuchten Konzepte einer Lösung von Integrationsproblematiken bei sprachübergreifender Simulation dienen. Durch zunehmenden Einsatz der Metamodellierung soll dabei (a) Vergleichbarkeit von Verhalten verschiedener Sprachen, (b) das Einbeziehen von Teilen echter Systeme in die Simulation ohne proprietäre Schnittstellen und Brüche in der Beschreibung operationaler Semantik und (c) das Sammeln von Testdaten während der Simulation zur Wiederverwendung in Test- und Integrationsphasen ermöglicht werden.

1.4 Lösungsansatz und Aufbau der Arbeit

Der metamodellbasierte Sprachentwurf ermöglicht eine nach objektorientierten Prinzipien strukturierte, universelle Vorgehensweise und wird deshalb als Basis zur Modell- und Sprachdefinition verwendet. In diesem Kontext wird darauf aufbauend mittels MActions eine Sprache zur operationalen Semantik entwickelt, die sich durch Erweiterungen von MOF in die Metamodelle nahtlos einfügt und eine Modellausführung und -analyse erlaubt. Für dieses Framework werden mittels Zustandsbetrachtungen

von Ausführungsläufen dynamische Analysen und Verhaltensvergleiche metamodellunabhängig möglich. Dieser Linie folgend gliedert sich die Dissertation im Weiteren in die folgenden Kapitel:

- In Kapitel 2 werden die Grundlagen eingeführt, auf denen die Forschungsarbeit fußt. Neben Basisdefinitionen wird der Metamodellierungsstandard MOF um ein neues Instanziierungskonzept ergänzt, mit dessen Hilfe sich Laufzeitstrukturen für operationale Semantiken integrieren lassen. Das dadurch entstehende εMOF- Framework wird als semantische Referenz der weiteren Arbeit algebraisch formalisiert (Abschn. 2.4). Anschließend führen wir die etablierten Kalküle SOS und ASM zu operationaler Semantik ein, um darauf aufbauend die Aktionssemantik MActions definieren zu können. Das Kapitel schließt mit einem Rückblick auf Konzepte der Temporallogik, welche für die dynamische Analyse in Kapitel 4 mittels LT-OCL essentiell sind.
- Das entstandene Metamodellierungs- und Simulationsframework der MActions wird in Kapitel 3 definiert. Dort findet sich die metazirkuläre Definition der Sprache für MOF-Modelle, welche dadurch eine operationale Semantik erhalten und weitergehend analysiert werden können. Zusätzlich wird durch Rückgriff auf die Referenzsemantik aus Kapitel 2 eine äquivalente Sprachdefinition angegeben, die zum einen als formale Basis und zum anderen in Kapitel 5 der Diskussion der Unterschiede im Vergleich zu etablierten Ansätzen dient. Das Kapitel schließt mit dem Beweis der Turing-Vollständigkeit der MActions.
- Kapitel 4 entwickelt das Framework weiter und zeigt, wie Modellzustände und deren Aufzeichnung zu einer dynamischen Analyse der Modelle genutzt werden. Basierend auf diesen Zustandsdefinitionen wird gezeigt, wie der Verhaltensvergleich von Modellen im Sinne einer Bisimulation möglich gemacht wird. Zur automatisierten Auswertung wird die Sprache LT-OCL entworfen und anhand einer Fallstudie beispielhaft deren Anwendung gezeigt. Eine weitere Fallstudie zur Sprache C# wurde aus Gründen der Übersichtlichkeit in Appendix A verlagert.
- Eine kritische Reflexion der Konzepte und Ergebnisse dieser Arbeit findet sich in Kapitel 5. Die vorangegangenen Kapitel vermeiden eine detaillierte Auseinandersetzung zum Vorteile einer klaren und kompakten Darstellung. Es werden neben einer Gegenüberstellung der Kalküle SOS, ASM und MActions u.a. die Motivation des Ansatzes reflektiert, Teilaspekte zur operationalen Semantik diskutiert und in Relation zu verwandten Arbeiten gesetzt. Offengebliebene Fragestellungen und eine mögliche Perspektive zur weiteren Forschung in Richtung modellorientierter Analysen runden das Kapitel ab, bevor eine Kapitel 6 die wichtigsten Ergebnisse zusammenfasst.

KAPITEL 2

Grundlagen, Formalisierung, Methoden

In diesem Kapitel werden Grundlagen erarbeitet, die im Weiteren als Basis zur operationalen Semantik in Metamodellen und der dynamischen Analyse dienen. Dazu wird in einem ersten Schritt die Metamodellierung konzeptionell erweitert und mathematisch formalisiert. Das entstehende εMOF erweitert dabei das MOF-Meta-Metamodell um das Konzept einer logischen Instanziierung und wird mittels universeller Algebra formal gefasst. Darauf aufbauend lässt sich im Anschluss für die operationale Semantik eine Aktionssemantik definieren, die als Ausführungskalkül mittels einer Erweiterung der *Abstract State Machines* (ASM) beschrieben ihrerseits semantisch präzise definiert wird. Dazu ist ein spezieller ASM-Typ nötig, der die Konzepte der Metamodellierung mit Metaebenen und Instanziierung sowie OCL verbindet und als formaler Kern der MActions und der dynamischen Analyse dienen wird.

2.1 Metamodellierung

Die Metamodellierungsarchitektur MOF mit ihrem Schichtenmodell der Metaebenen bildet die konzeptionelle Referenz zur objektorientierten Metamodellierung[1]. Es existieren verschiedene Implementierungen des Standards, die die plattformunabhängigen Konzepte in unterschiedlichen Technologien realisieren (z.B. [72][73][74][70]). In dieser Arbeit beziehen wir uns auf das *Eclipse Modeling Framework* (EMF) als Quasi-Referenzimplementierung[2] des EMOF-Teils des Standards [72], welches als Basis für die prototypische Umsetzung dieser Arbeit diente. Die technologiebedingten Abweichungen zur MOF-Spezifikation sind an gegebener Stelle kenntlich gemacht und diskutiert.

Aus diesem Grund werden wir zunächst die MOF-Spezifikation bzgl. Metaebenen genauer untersuchen und anschließend in Abschnitt 2.4.2 eine mathematische Fundierung der Konzepte erarbeiten.

[1] Als Referenz dient uns die *Meta Object Facility* (MOF) Version 2.x, da alle wesentlichen OO-Konzepte enthalten sind, vgl. [8]. Für eine allgemeine Definition zu 'Objektorientierten-Metamodellierungsframeworks' s.a. Scheidgen [71]

[2] vgl. OMG/Eclipse Symposium [75]

2.2 MOF-Metamodellierungsarchitektur

Kern der MOF-Spezifikation ist die Schichtenarchitektur, bei der Modelle anhand ihrer gegenseitigen Beziehung *Metaebenen* zugeordnet werden (vgl. [76]). Dabei ist ein Modell einer Ebene immer über ein Metamodell der darüber liegenden Ebene strukturell beschrieben.[3] Während die MOF1.x-Spezifikationen noch von den klassischen vier Schichten M0 bis M3 ausgehen, verallgemeinert die MOF2.0 diese Architektur auf beliebig viele Ebenen und rückt die Beziehung zwischen Objekt (Daten) und Klasse (Metadaten) in den Mittelpunkt. Entscheidend ist, dass die Eigenschaften eines jeden Objekts durch Navigation zu seinem beschreibenden *Metaobjekt* (Classifier) abgefragt werden kann (vgl. Abschnitt 7.2. in [8]).

Die Thematik der Instanziierung bekommt besondere Relevanz für diese Arbeit, wenn man sich der dynamischen Semantik in Bezug auf die Beschreibung des Laufzeitmodells nähert. Wie einleitend erwähnt, beschreibt ein Metamodell die abstrakte Syntax einer Sprache. Inhaltlich enthält dieses demnach nur diejenigen Sprachmittel, die eine Sprache rein strukturell ausmachen (also quasi aus Sicht des Anwenders). Werkzeuge, die das dynamische Verhalten realisieren, müssen darüber hinaus ein Laufzeitmodell implementieren, welches implizit in der informell beschriebenen dynamischen Semantik vorgegeben ist. So ist z.B. durch Quelltext der Programmiersprache C – also einem Modell in Textform – ein Verhalten definiert, welches durch Ausführung auf einer oder mehrerer Maschinen umgesetzt wird. Diese Maschinen verwalten die Laufzeitgrößen in strukturierter Form, die durch die Ausführungssemantik der Sprache C im Standard festgehalten sind. Für dieses Beispiel sind das Programmschrittzähler, Variablenbelegungen, allozierter Heap, Stackframes etc. Um eine operationale Semantik zu beschreiben, fehlen im Metamodell also Konzepte für Laufzeitgrößen und deren Verhalten. Folgt man der Metapher von Programm und Prozess, so kann ein konkretes Prozessverhalten als *Instanz eines Programms* betrachtet werden. Als Folge lassen sich deshalb Laufzeitgrößen in die Schichtenarchitektur als „M0-Laufzeitstrukturen“ einordnen. Die Beziehungen zwischen den Modellen ist in Abbildung 2.1 dargestellt.

Konzeptionell verbindet die Objekte auf den Metaebenen M1 bis M3 die *instanceOf*-Beziehung. Im Zentrum steht dabei das Metamodell als abstrakte Syntax, das um Elemente für Laufzeitgrößen erweitert wird. Dieses zweite Metamodell definiert die Struktur der Laufzeitgrößen als Laufzeit*metamodell* und ist selbst eine „echte“ Instanz des MOF-Meta-Metamodells. Bereits eingezeichnet ist auch die Ergänzung der MActions auf M3, welche Konstrukte zur Definition der operationalen Semantik ermöglicht. Die Beziehungen zwischen den Modellen auf M2 sowie die MActions werden im Kapitel 3 genauer beschrieben. Im Folgenden soll zunächst ein genauerer Blick auf die Einordnung des Laufzeitmodells geworfen werden.

Die Laufzeitumgebung transformiert das Laufzeitmodell während der Ausführung anhand einer operationalen Semantik (M2) und erzeugt dadurch das eigentliche Verhalten. Konsequenterweise kann demnach aus Sicht des M1-Modells das Laufzeitmodell wie oben erwähnt der Ebene M0 zugeschrieben werden, jedoch ist es als Instanz des Laufzeitmetamodells (M2) strukturell beschrieben.

Dieser Widerspruch lässt sich ohne weiteres nicht auflösen, es sei denn, man verzichtet auf die Instanzbeziehung zwischen M1-Modell und Laufzeitmodell, oder erlaubt einen „Bruch“ zwischen den Ebenen. Aus dieser Argumentation heraus bedarf es einer genaueren Betrachtung der Instanziierung und der mit ihr verbundenen

[3] Generell spricht man von Modellen der Ebene Mx, um die Relationen zwischen Modellen klar zum Ausdruck zu bringen

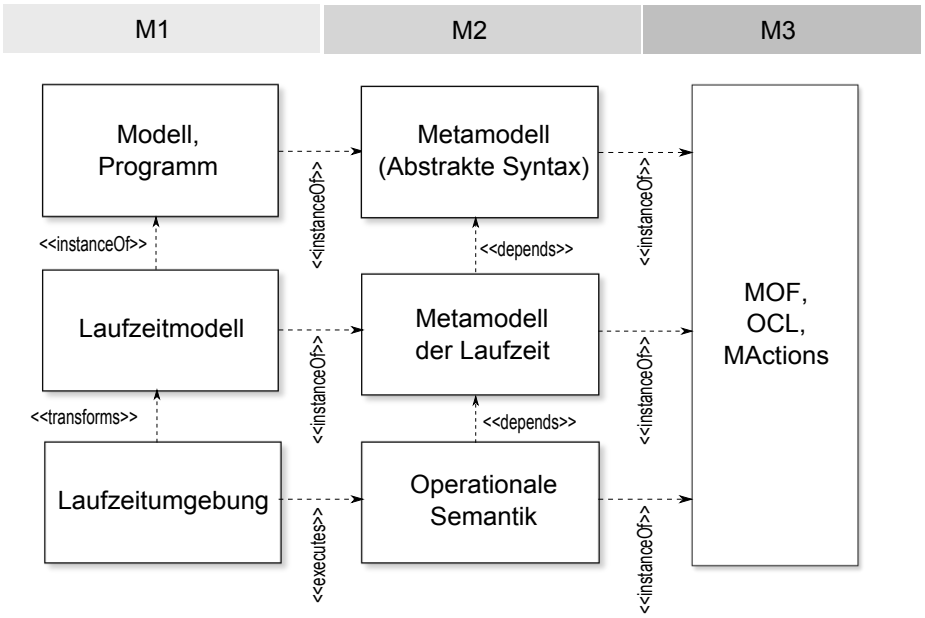


Abbildung 2.1: Metaebenen

Probleme.

2.3 Instanziierung und Metaebenen

Tatsächlich ist die Metadatenhierarchie durch das `Reflection Package` im MOF-Standard spezifiziert (vgl. [8], Kapitel 9). Die Klasse *Object*, als Basistyp für alle Instanzen, erlaubt die Navigation zwischen Metaebenen mit Hilfe der Operation `getMetaClass`, die diejenige Klasse zurückliefert, welche Attribute und Operationen des Objekts definiert. Dabei kann diese Klasse durch ihre Eigenschaft als *Objekt* wiederum erneut Instanz einer anderen Metaklasse sein. Den Abschluss in den Ebenen bildet die meta-zirkuläre Definition ≪der einen≫ MOF-Klasse $\textsc{Class}_{M3}$ und somit des gesamten Meta-Metamodells über sich selbst.

Die im MOF-Standard verwendete Form der Instanziierung beruht auf der zweistufigen Unterscheidung zwischen Typebene (Klasse) und Instanzebene (Objekt), welche auch als *flache* Instanziierung (*engl.: shallow instantiation*) bezeichnet wird. Wie in objektorientierten Programmiersprachen spezifiziert die Klasse die Eigenschaften, die die Instanzen sowohl in ihrer Struktur als auch in ihrem Verhalten charakterisieren. Die strukturellen Eigenschaften von MOF-Instanzen sind im *MOF Instance Model* beschrieben (s. Abbildung 2.2). Instanzen besitzen für Attribute ihrer Klasse sog. *Slots*, die Werte für primitive Datentypen (*StructureSlot*) oder Referenzen (*LinkSlot*) enthalten.

In Bezug auf die Zuordnung zu den Metaebenen ergibt sich allerdings aus dieser Instanziierung eine Dichotomie, die der MOF-Standard nicht auflöst: während eine Klasse *Meta-A* auf Ebene M2 als Instanz *der* MOF-Klasse $\textsc{Class}_{M3}$ die Struktur seiner Instanzen auf M1 klar festlegt, bezeichnen wir diese mit A, so ist nicht genauer definiert wie die Instanzen von A auf M0 wiederum strukturiert sind. Angenommen A ist auf M1 nun wiederum eine Klasse (Instanz der Klasse $\textsc{Class}_{M3}$), stellt sich die Frage, ob nun *Meta-A* seine Struktur beschreibt oder aber $\textsc{Class}_{M3}$. Es bleibt unklar, welche Slots A besitzt. So gesehen ergibt sich ein Widerspruch, wenn man

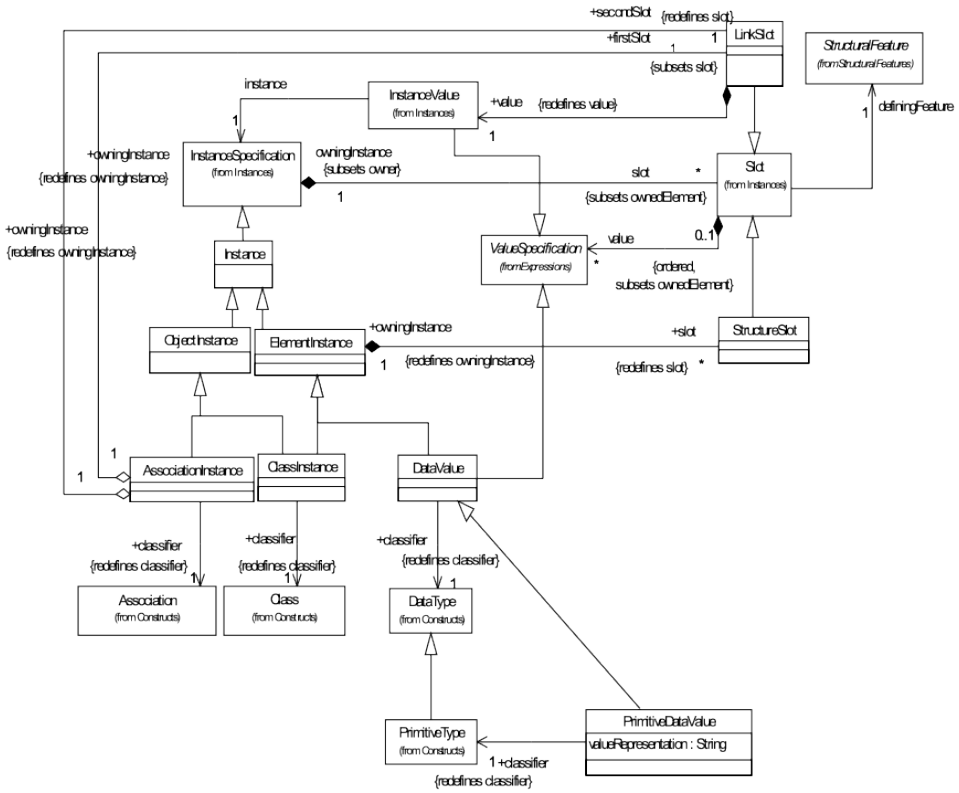


Abbildung 2.2: CMOF Instance Model (Quelle: Abschnitt 15 *CMOF Abstract Semantics*, [8])

annimmt, dass A nun *beides* ist, eine Instanz von *Meta-A* (mit entsprechenden Eigenschaften) und Class_{M3}. Andersherum folgt daraus, dass MOF nicht mit mehr als den drei Ebenen M1 bis M3 umgehen kann.

Schaut man sich Implementierungen des Standards an, wie diese in der Praxis das Problem lösen, so stellt man fest, dass alle genau auf drei Metaebenen beruhen [72][77][73][12][78].

2.3.1 Logische Instanziierung

Die Ursache des obigen Problems liegt in der Verbindung der Definition von flacher Instanziierung mit der Zuordnung von Objekten zu Metaebenen. Da durch die flache Instanziierung von Klasse zu Objekt letztere keine Instanzen haben, läuft die Instanziierung nach zwei Ebenen zwangsläufig aus, d.h. MOF (Ebene M3) erlaubt die Definition von Klassen (Ebene M2), die wiederum Objekte beschreiben (Ebene M1). Klar definiert ist dabei die Instanziierung M3-M2 sowie M2-M1, weil diese per se durch die flache Instanziierung vorgegeben ist, also jede Klasse wird instanziiert zu einem Objekt diesen Typs, jedes Attribut/Referenz der Klasse zu Slots innerhalb des Objekts etc. Im Gegensatz dazu existiert kein Mechanismus, der die „Instanziierung von M1-Objekten“ regelt (vgl. Widerspruch in Abschnitt 2.3). Anders gesagt gibt es in MOF keine Möglichkeit auszudrücken, ob und wie Objekte die Instanziierungseigenschaft erhalten, außer wenn sie eine direkte Instanz von Class_{M3} sind.

Zur Lösung dieses Problems wurden in der Vergangenheit verschiedene theoretische Vorschläge diskutiert, um die Multiebenen-Metamodellierung zu ermöglichen. Diese reichen von *PowerTypes* und des *Prototypical Concept Patterns*, die eine

Verbindung von Vererbung und Instanziierung verwenden, bis hin zu *tiefer* Instanziierung (*engl. deep instantiation*). Für eine Übersicht und Zusammenfassung der Vorschläge siehe Atkinson und Kühne in [79][80].

Nach einer Reihe von Konzeptversuchen entlang kleinerer Beispielsprachen wurde untersucht, inwieweit sich diese Vorschläge zur Anwendung auf die Definition des Laufzeitmodells eignen. Dabei konnten die folgenden Beobachtungen gemacht werden:

- Die Modellierung der Laufzeitinstanzen auf M0 folgte nie strikt der Instanziierungsbeziehung. Strukturell haben M1-Objekte also nicht als 'Classifier' die Struktur der M0-Objekte definiert. Die notwendigen Informationen der Laufzeitzustände beschränkte sich überwiegend auf Ergänzungen der im M1-Modell enthaltenen Strukturen um Zeiger zur Ablaufbeschreibung und Strukturen zur Verwaltung von Werten bzw. Zuständen.
- Obwohl verschiedene Sprachen modelliert wurden, folgte die Zuordnung von M0-Objekten zu M1-Objekten gleichbleibend dem kontextsensitiven Muster der Instanzbeziehung, bei dem ein zugeordnetes M1-Objekt eine Art Bezeichner für eine Gruppe von Laufzeitobjekten darstellt.
- Auch wenn die Trennung zwischen abstrakter Syntax und Laufzeitmodell konzeptionell eine Zuordnung zu verschiedenen Metaebenen rechtfertigt, lässt sich selbiges Ziel auch ohne neue Sprachkonzepte erreichen, indem alle Laufzeitkonzepte einfach durch die normalen OO-Sprachkonstrukte im selben Metamodell modelliert werden.
- Die benötigten Laufzeitstrukturen konzentrieren sich im Vergleich zur abstrakten Syntax auf einige wenige Konstrukte. Bei der expliziten Erweiterung des Metamodells um diese Laufzeitstrukturen wurden selbst bei der Modellierung einer Maschine zur Definition der Programmiersprache C# (inkl. Threading) weniger als ein Dutzend Klassen benötigt, um alle Laufzeitkonzepte zu erfassen (vgl. Appendix A)

Unter Berücksichtigung dieser Tatsachen wurden publizierte Vorschläge zur Multiebenen-Metamodellierung mit den aufgeführten Beobachtungen verglichen und auf ihre Anwendbarkeit hin bewertet. Da keiner dieser Vorschläge alle Anforderungen erfüllte, ergab sich für uns die Notwendigkeit, ein erweitertes Instanziierungskonzept zu entwerfen, welches wir als *logische Instanziierung* bezeichnen.

Kernidee ist die Differenzierung der gängigen, flachen Instanziierung in der Objektorientierung zum Erzeugen eines Objekts mit der in der Klasse vorgegebenen Struktur (d.h. Slots für Attribute und Referenzen) von der rein logischen *Metaobjekt*- Referenzierung. Erstere bezeichnen wir deshalb auch als *physikalische* instance-of (PIO)-Relation, die neben der Struktur eines Objekts auch seinen Typ (Classifier) festlegt, der für Inklusionspolymorphie von Attributen und Operationen ausgewertet wird. Im Gegensatz dazu bietet die logische instance-of (LIO)-Relation eine an jedem Objekt verfügbare Referenz zu einem anderen Objekt: seinem *Metaobjekt*. Diese Relation stellt eine besondere Form der Assoziation dar, welche während der Erzeugung einmalig auf das Metaobjekt gesetzt wird. Ab diesem Zeitpunkt ist eine Navigation zwischen Objekt und Metaobjekt möglich (und ändert sich nicht mehr).

2.3.2 Definition der Instanziierung

Um die erläuterten Formen der Instanziierung zu unterscheiden, geben wir zunächst ein eigenes Instanzmodell für MOF an, welches das in Abb. 2.2 gezeigte ersetzt und

beide Formen der Instanziierung (PIO und LIO) unterstützt. Dieses wurde in vereinfachter Form bereits in [69] zur Coevolution von Modellen und Metamodellen verwendet. Zentrale Bestandteile sind die Klassen `Object` und `Slot`, welche *orthogonal* zum MOF-Meta-Metamodell Objektstrukturen beschreiben. Wohlgemerkt wollen wir nicht von der flachen Instanziierung abrücken, sondern diese zunächst präzise beschreiben.

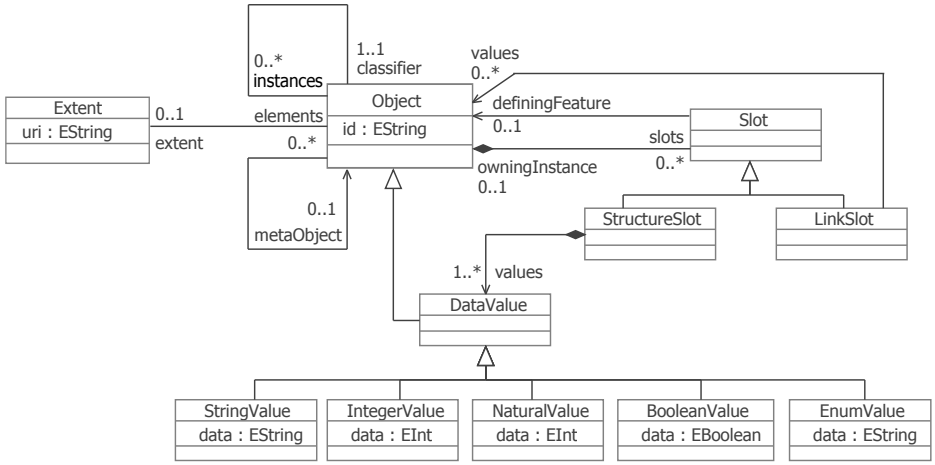


Abbildung 2.3: Instanzmodell

Löst man sich für einen Moment von der Zuordnung eines Modells oder Objekts zu einer Metaebene, so kann der gesamte Modellraum als ein zusammenhängender Objektgraph angesehen werden, bei dem einigen Relationen eine besondere Bedeutung zukommt. Instanziierung ist so gesehen eine geregelte Form der Erweiterung dieses Graphen, indem man an die Instanziierung gewisse strukturelle Bedingungen knüpft, wie neue Objekte beschaffen sein sollen. Diese reinen Struktureigenschaften regelt MOF letztendlich informell, in dem es sich auf die Klasse-Objekt-Beziehung (und damit der gängigen, flachen Instanziierung) stützt. Mittels des Instanzmodells aus Abb. 2.3 können wir den gesamten Objektgraphen nachbilden und diese Instanziierungsbeziehung deklarativ explizit formulieren. Dabei ist die Klasse-Objekt-Beziehung durch die Referenz `classifier` repräsentiert. Für die Klasse `Object` ergeben sich folgende Bedingungen:

```
context Object
inv C01:
-- Nur die Klasse EMOF::Class im MOF-Modell kann zweimal instantiiert werden
self.classifier.classifier = EMOF::Class

inv C02:
-- für alle Properties einer Metaklasse und ihrer Superklassen existiert
-- ein entsprechender Slot in der Instanz
let
  metaProperties = self.classifier.collectSupertypes->collect
  ( slots->select ( prop.definingFeature = EMOF::ownedProperties).values)
in
  metaProperties->forAll( attr | self.slots.exists(s|s.definingFeature = attr))
and
  metaProperties->size() = slots.size()

-- Hilfsoperation zum Sammeln aller Superklassen
```

```
def collectSupertypes : Set(Object) =
  let supertypes = self.slots.select(definingFeature = EMOF::superClass) in
      supertypes.collect(collectSupertypes).union(supertypes).union(self)
```

Durch `C01` und `C02` ist sowohl die Struktur als auch die Beziehung einer Instanz zur definierenden Klasse festgelegt. Der rekursive Abschluss über die Klasse EMOF::Class kann ferner durch spezielles Setzen der `classifier`-Referenz dieser Klasse auf sich selbst (d.h. `self`) beschrieben werden. Intuitiv ergibt sich für jedes Property (d.h. Attribute und Referenzen) ein entsprechender Slot, der Informationen aufnehmen kann. Welche Informationen in einem Slot abgelegt werden können, ist durch die folgenden Konsistenzbedingungen der Klasse `Slot` geregelt:

```
context Slot
inv C03:
-- Der classifier eines Slots ist immer EMOF::Property
definingFeature.classifier = EMOF::Property

inv C04:
-- Werte in Slots haben den Typ den das Property spezifiziert
let
  type = definingFeature.slots.select(definingFeature = EMOF::type).values
in
  type.notEmpty() implies values->forAll(classifier = type)

inv C05:
-- Kardinalitäten der Werte entsprechend der Spezifikation ihres Classifiers
let
  lower = definingFeature.slots.select(definingFeature = EMOF::lower).values
  upper = definingFeature.slots.select(definingFeature = EMOF::upper).values
in
  values->size() > lower and values->size() < upper

inv C06:
-- Mengeneigenschaften isOrdered und isUnique sind ebenfalls passend zur
-- Spezifikation des Properties
let
  isUnique = definingFeature.slots.select(definingFeature = EMOF::isUnique)
in
  isUnique implies values.isUnique()
```

Das Constraint `C03` korrespondiert zu `C01` um sicherzustellen, dass die Instanziierung von Properties immer in Slots entsprechender Instanzen mündet. Die restlichen Bedingungen `C04` bis `C06` garantieren die weitergehenden Eigenschaften eines Attributs/einer Referenz.

Zurückkommend auf die Diskussion der Metaebenen kann die Ebene, auf der sich ein Objekt befindet, a posteriori anhand seiner `classifier`-Referenz nur *relativ* bestimmt werden. Legt man die MOF-Klasse als quasi Fixpunkt mit M3 fest, können demnach alle Objekte eines Metamodells für die gilt `self.classifier = EMOF::Class` als M2 zugehörig betrachtet werden, während M1-Objekte als Instanzen dieser M2-Klassen nur der obigen Bedingung `C01` genügen. Trotzdem ist Vorsicht geboten. Mit gleicher Argumentation können auch die Klassen des MOF-Modells selbst als Instanzen von EMOF::Class als M2 zugehörig betrachtet werden! Dieser (scheinbare) Widerspruch löst sich auf, wenn man bedenkt, dass die Klassen im MOF-Modell meta-zirkulär beschrieben sind: MOF ist eine Instanz von sich selbst. Andersherum gesehen existieren eigentlich immer nur zwei „echte“ Metaebenen (sieht man von EMOF::Class einmal ab), nämlich die der Klassen (M2) und die

ihrer Instanzen (M1), weil wir dies ja auch als flache Instanziierung gefordert haben.

Mit Blick auf die in Abschnitt 2.3.1 eingeführte logische Instanziierung, lässt sich MOF um eine zusätzliche Referenz *metaClass* erweitern, die dieser Relation zwischen Klassen entspricht. Abbildung 2.4 zeigt den relevanten Auszug des MOF-Modells mit der neuen Referenz.

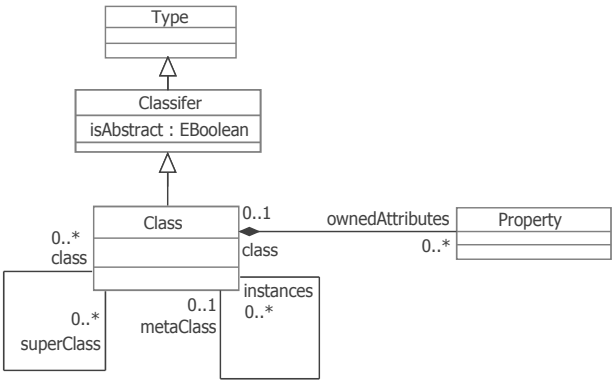


Abbildung 2.4: Um logische Instanziierung erweitertes MOF-Modell (Auszug)

Für das Instanzmodell ergibt sich eine entsprechende Erweiterung um die Referenz `metaObject`, die bereits in Abb. 2.3 verzeichnet ist. Das Verhältnis zwischen beiden im Objektgraphen beschreiben wir mit einer ergänzenden Bedingung:

context Object
inv C07:
−− Stehen zwei Klassen in einer logischen Instanziierungsbeziehung, so müssen Instanzen
−− dieser Klassen entsprechend mit einer metaObject−Referenz versehen sein
let
  metaClass = self.classifier.slots.select(definingFeature = EMOF::metaClass).values
in
  metaClass **implies** self.metaObject.classifier = metaClass

Es sei angemerkt, dass wir im MOF-Modell nur die Referenz `metaClass` eingeführt haben, diese allerdings nicht selbst auf das MOF-Modell angewendet haben. Da im Prinzip alle Klassen *physikalische* Instanzen von `MOF::Class` sind, könnte also lediglich `MOF::Class` eine logische Instanz von sich selbst sein, da alle anderen Klassen keine Objekte als Instanzen besitzen. Faktisch führt dies im Objektgraphen allerdings dazu, dass alle Klassen in einem Metamodell zusätzlich zur `classifier`-Referenz auch noch eine `metaObject`-Referenz auf `MOF::Class` hätten. Sinnvoller ist es, das Instanzmodell als logisches „Laufzeitmodell“ für ein Metamodell zu betrachten. Deshalb kann die Klasse `Object` als Instanz von `MOF::Class` als alternative Sicht auf den Modellgraphen gesehen werden. Beide beschreiben unterschiedliche Aspekte der Metadatenhierarchie, tatsächlich müssen wir uns aber für eine Form der Modellierung entscheiden (vgl. 2.4.2).

2.3.3 Beispiel zur logischen Instanziierung

Die eingeführte logische Instanziierung soll ein Beispiel verdeutlichten. Abbildung 2.5 zeigt ein M2-Metamodell mit drei Hauptklassen `MetaConcept`, `Concept` und `Instance`, die durch die logische InstanceOf-Beziehung strukturell Konzepte über drei logische Metaebenen LM1 bis LM3 hinweg beschreiben. Während `MetaConcept` einen Namen hat, bestehen `Concept`-Objekte aus einer Menge von `Items`. Instanzen von `Concept`, modelliert durch die Klasse `Instance`, besitzen nur einen Wert.

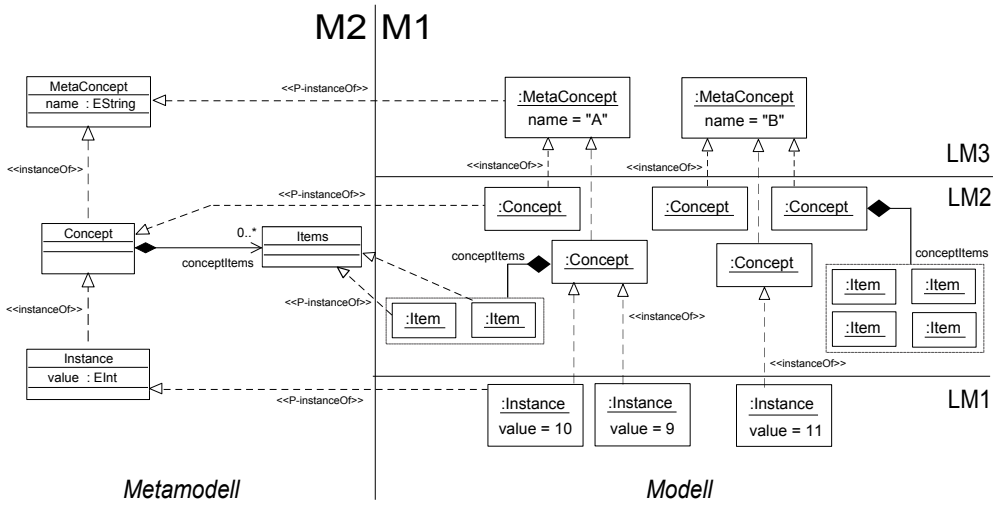


Abbildung 2.5: Metamodell mit mehreren logischen Metaebenen

Als physikalische Instanzen (`<<P-instanceOf>>`, `classifier` Link) sind z.B. die als M1-Modell kenntlich gemachten Beispielinstanzen beschrieben, die über drei logische Metaebenen verteilt sind. Untereinander sind die Objekte mittels der logischen `instanceOf`-Beziehung verbunden (also durch den `metaObject`-Link). Somit kann ein Metamodell wie gezeigt beliebig viele Metaebenen definieren, wobei es selbst die Struktur aller Instanzen beschreibt.

2.4 Formalisierung von Metamodellen

Unsere Strategie zur mathematischen Formalisierung setzt auf den algebraischen Beschreibungsformen der Signaturen und Algebren auf. Diese erlauben eine datenstrukturorientierte Sicht auf Modelle und versprechen eine abstrakt kompaktere Beschreibungsform als elementarere mathematische Strukturen. Weitere Argumente sind die bereits existierende Formalisierung der Sprache OCL mittels einer Algebra und nicht zuletzt die Aussicht, eine vermeintlich native Verbindung mit ASM herstellen zu können. Aus diesen Gründen wurden Alternativen wie z.B. attributierte Graphen, Kategorien oder andere Strukturen verworfen. Aktuelle Forschungen zu MOF untersuchen vergleichbare Ansätze, z.B. durch eine Abbildung auf ausführbare Termersetzungslogiken (engl. *executable rewriting logic*, vgl. [81]), die im Kern wiederum auf Algebren basieren.

2.4.1 Grundlagen: Algebren

Dieser Abschnitt führt die wichtigsten algebraischen Definitionen und Notationen ein. Wo immer möglich folgen wir den gängigen Notationen (s.a. [82][83]).

Definition 1 (Signatur) *Sei $\mathcal{S}$ eine nichtleere Menge von Sorten und Ω eine Familie von n-stelligen Operationssignaturen der Form $\Omega = (\Omega)_{op}$ mit $op \in (\mathcal{D}_1 \times \cdots \times \mathcal{D}_n \times \mathcal{C})^*$, wobei $\mathcal{D}_{\{1..n\}}, \mathcal{C} \in \mathcal{S}$. Dann bezeichnet das Tupel $\Sigma = (\mathcal{S}, \Omega)$ eine algebraische* Signatur. *$\mathcal{D}$ bezeichnet die Domänen einer Operation, $\mathcal{C}$ die Kodomäne. Einstellige Operationen bezeichnen wir als Konstanten.*

Ergänzend zu Signaturen nutzen wir Familien von freien Variablen $Var = (Var_s)_{s \in \mathcal{S}}$, meist um Substitutionen in Termen durchzuführen. Wir verzichten auf separate strikte Definition einer Signatur mit Variablen (etwa $\Sigma = (\mathcal{S}, \Omega, Var)$) und verwenden Variablenmengen freizügig. Für Variablen fordern wir o.B.d.A. eine Benennung, die nicht mit den Operationen kollidiert und eindeutig ist.

Definition 2 (Σ-Algebra) *Sei $\Sigma = (\mathcal{S}, \Omega)$ eine Signatur gemäß Definition 1. Für alle $op : s_1 \dots s_n \to s \in \Omega$ sei $op_{\mathbb{A}} : A_{s_1} \times \cdots \times A_{s_n} \to A_s$ eine Abbildung über $A_s, s \in \mathcal{S}$ als Mengen. Dann bezeichnet das Paar $\mathbb{A}_\Sigma = ((A_s)_{s \in \mathcal{S}}, (op_{\mathbb{A}})_{op \in \Omega})$ eine* Σ-Algebra. *A_s nennen wir die Trägermenge zur Sorte s, $op_{\mathbb{A}}$ die Interpretation von op in $\mathbb{A}$. Für Konstantensymbole $c :\to s$ fordern wir, dass ein ausgezeichnetes Element $c_{\mathbb{A}} \in A_s$ definiert wird.*

Eine Algebra bezeichnen wir auch als *semantische Domäne* einer Signatur, frei nach Scott-Strachey (vgl. [84]). Als Syntaxkonvention notieren wir Trägermengen (*engl. domains*) als Menge in Großbuchstaben der Form:

OBJECTS = { Menge aller Objekte }
BOOLEAN = $\{true, false\}$
STRING = $\{\omega \mid \omega \in \mathcal{N}+\}$

Definition 3 (Terme) *Die sortenindizierte Familie von Mengen $T_{\Sigma,s}(Var)$ über Variablen Var zur Sorte $s \in S$ bezeichnet (allgemeine)* Terme *über der Signatur Σ und ist wie folgt definiert:*

1. *Variablen sind Terme: $v \in T_{\Sigma,s}(Var)$ für alle $v \in Var_s$*
2. *Konstanten sind Terme: $c \in T_{\Sigma,s}(Var)$ für alle $c :\to \mathcal{C}$ mit $c \in \Omega$*

3. *Operationen sind Terme:* $op(t_1, \ldots, t_n) \in T_{\Sigma,s}(Var)$ *für alle* $op : s_1 \ldots s_n \to s \in \Omega$ *und* $t_{1..n} \in T_{\Sigma,s}(Var)$

Ein Term ist geschlossen, *wenn er keine Variablen enthält, andernfalls ist er* offen.

Terme über einer Signatur lassen sich unter Angabe einer Variablenbelegung und einer Algebra *interpretieren*. Für die Variablenbelegung nehmen wir eine Abbildungsfamilie β an, mit $\beta_s : Var_s \to A_s$ zur Sorte s, die jeder Variablen ein Element aus A_s zuweist. Der Einfachheit halber lassen wir die Indizes weg, wenn sich die Zuordnung aus dem Kontext ergibt.

Definition 4 (Termauswertung, Terminterpretation) *Sei Σ eine Signatur, $\mathbb{A}_\Sigma$ eine Algebra und β eine Variablenbelegung. Die Termauswertung (syn.* Interpretationsfunktion*) $I : T_\Sigma(Var) \to op_\mathbb{A}$ ist eine Familie von Abbildungen für alle Sorten $s \in \mathcal{S}$, die sich rekursiv definiert:*

1. *für alle Variablen* $v \in Var_s : I_s[\![v]\!] := \beta_s(v)$
2. *für alle Konstanten* $c :\to s \in \Omega : I_s[\![c]\!] := c_\mathbb{A}$
3. *für alle Terme* $op(t_1, \ldots, t_n) \in T_{\Sigma,s}(Var)$: $I_s[\![op(t_1, \ldots, t_n)]\!] := op_\mathbb{A}(I[\![t_1]\!], \ldots, I[\![t_n]\!])$

Durch Formeln, Gleichungen und Prädikate werden Σ-Algebren im Allgemeinen um mehrsortige Prädikatenlogik erster Stufe ergänzt:

Definition 5 (Formeln, Prädikate, Gleichungen) *Sei $T_\Sigma(Var)$ die Menge allgemeiner Terme zur Signatur Σ, dann bezeichnet die Menge $F_\Sigma(Var)$ die* Formeln *über Σ und der Variablenmenge Var:*

i. Verum und Falsum sind Formeln: $\top, \bot \in F_\Sigma(Var)$
ii. Propositionen sind Formeln: $\neg\varphi, (\varphi \vee \psi), (\varphi \wedge \psi), (\varphi \to \psi), (\varphi \leftrightarrow \psi) \in F_\Sigma(Var)$, *für* $\varphi, \psi \in F_\Sigma(Var)$
iii. Gleichungen sind Formeln: $(s = t) \in F_\Sigma(Var)$, *mit* $s, t \in T_\Sigma(Var)$
iv. Prädikate sind Formeln: $p(t_1, ..., t_n) \in F_\Sigma(Var)$ *wobei* $p : \langle s_1 ... s_n \rangle \in P$ *eine Menge von n-stelligen Prädikaten ergänzend zu Σ ist und* $t_i \in T_\Sigma$ *für* $i = 1, \ldots, n$
v. Allquantor und Existenzquantor bilden Formeln: $(\forall x \varphi) \in F_\Sigma(Var)$ *und* $(\exists x \varphi) \in F_\Sigma(Var)$, *für* $\varphi, \psi \in F_\Sigma(Var)$ *und* $x \in Var$

Die Menge der durch Quantoren gebundenen Variablen einer Formel (oder Terme) $\varphi \in F_\Sigma(Var)$ bezeichnen wir mit $BV(\varphi)$, freie Variablen mit $FV(\varphi)$. Im Gegensatz zu Elementen der Trägermengen einer Algebra erhalten Formeln immer einen Wahrheitswert bei der Auswertung:

Definition 6 (Formelauswertung, Gültigkeit) *Seien $\varphi, \psi \in F_\Sigma(Var)$ Formeln über einer Signatur Σ, $\mathbb{A}_\Sigma$ eine Algebra und β eine Variablenbelegung. Die Gültigkeit einer Formeln φ ist die Zuordnung eines Wahrheitswertes, geschrieben $[\![\varphi]\!]^\mathbb{A}_\beta \in \{true, false\}$, und induktiv definiert:*

1. $[\![\top]\!]^\mathbb{A}_\beta := true$
2. $[\![\bot]\!]^\mathbb{A}_\beta := false$
3. $[\![\neg\varphi]\!]^\mathbb{A}_\beta := \begin{cases} true & \text{wenn } [\![\varphi]\!]^\mathbb{A}_\beta = false \\ false & \text{sonst} \end{cases}$
4. $[\![\varphi \vee \psi]\!]^\mathbb{A}_\beta := \begin{cases} true & \text{wenn } [\![\varphi]\!]^\mathbb{A}_\beta = true \text{ oder } [\![\psi]\!]^\mathbb{A}_\beta = true \\ false & \text{sonst} \end{cases}$

5. $[\![\varphi \wedge \psi]\!]^{\mathbb{A}}_{\beta} := \begin{cases} true & \text{wenn } [\![\varphi]\!]^{\mathbb{A}}_{\beta} = true \text{ und } [\![\psi]\!]^{\mathbb{A}}_{\beta} = true \\ false & \text{sonst} \end{cases}$
6. $[\![\varphi \rightarrow \psi]\!]^{\mathbb{A}}_{\beta} := \begin{cases} true & \text{wenn } [\![\varphi]\!]^{\mathbb{A}}_{\beta} = false \text{ oder } [\![\psi]\!]^{\mathbb{A}}_{\beta} = true \\ false & \text{sonst} \end{cases}$
7. $[\![(s = t)]\!]^{\mathbb{A}}_{\beta} := \begin{cases} true & \text{wenn } [\![s]\!]^{\mathbb{A}}_{\beta} = [\![t]\!]^{\mathbb{A}}_{\beta} \\ false & \text{sonst} \end{cases}$
8. $[\![\forall x\varphi]\!]^{\mathbb{A}}_{\beta} := \begin{cases} true & \text{wenn für alle } a \in A_s \text{ gilt: } [\![\varphi]\!]^{\mathbb{A}}_{\beta[x/a]} = true \\ false & \text{sonst} \end{cases}$
9. $[\![\exists x\varphi]\!]^{\mathbb{A}}_{\beta} := \begin{cases} true & \text{wenn für mindestens ein } a \in A_s \text{ gilt: } [\![\varphi]\!]^{\mathbb{A}}_{\beta[x/a]} = true \\ false & \text{sonst} \end{cases}$

Eine Algebra $\mathbb{A}_\Sigma$ erfüllt *eine Formel unter der Belegung* β, *notiert* $(\mathbb{A}, \beta) \models \varphi$, *gdw.* $[\![\varphi]\!]^{\mathbb{A}}_{\beta} = true$, *sonst* $(\mathbb{A}, \beta) \not\models \varphi$.

Formeln und Prädikaten schaffen einen allgemeinen prädikatenlogischen Kern, der unabhängig von der „Werteinterpretation“ normaler Operationen semantisch gleichbleibend ist. Diesen verwenden wir in den weiteren Ausführungen nicht nur für die Definition von ASM, sondern auch um einheitlich über Temporallogik sprechen zu können (s. Abschnitt 2.7). Relationen und Prädikate verwenden wir synonym, d.h. ein Tupel der Form $r : \langle \mathcal{N}, \mathcal{N} \rangle = \{(x, y) \mid x \neq y, x, y \in \mathcal{N}\}$ bezeichnet ein Prädikat.

Anmerkung: Da Funktionen generell n-stellige Abbildungen der Form: $f : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \rightarrow \mathcal{C}$ sind, bezeichnet man auch ausgewiesene Funktionen wie: $p : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \rightarrow$ Boolean als Prädikate, d.h. Funktionen, die in die Menge Boolean $= \{true, false\}$ abbilden. Die Dualität zwischen Prädikaten als Teil von Formeln (ohne Zieldomäne) und expliziten Funktionen in die Domäne Boolean akzeptieren wir, damit prädikatenlogische Formeln eigenständig verwendet werden können.

Zudem werden wir gängige Datenstrukturen und damit einhergehend Syntaxerweiterungen verwenden, deren Definition hier kurz skizziert werden soll. Für Sequenzen von Werten führen wir einen Listentyp ein, der für jede Domäne $\mathcal{D}$ existiert (quasi als „Schablone“):

$Seq(\mathcal{D}) =_{def} \{\text{Index indizierte Multimenge über } \mathcal{D}\}$
$add(x, x_{Seq}) : \mathcal{D} \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$
$addAt(x, i, x_{Seq}) : \mathcal{D} \times \mathbb{N} \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$
$remove(x, x_{Seq}) : \mathcal{D} \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$
$removeAll(x_{Seq}, y_{Seq}) : Seq(\mathcal{D}) \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$
$first(x_{Seq}) : Seq(\mathcal{D}) \rightarrow \mathcal{D} \cup \{undef\}$
$select(x_{Seq}, i) : Seq(\mathcal{D}) \times \mathbb{N} \rightarrow \mathcal{D} \cup \{undef\}$
$empty(x_{Seq}) : Seq(\mathcal{D}) \rightarrow$ Boolean
$length(x_{Seq}) : Seq(\mathcal{D}) \rightarrow \mathbb{N}$
$\cup_{Seq} : Seq(\mathcal{D}) \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$
$\cap_{Seq} : Seq(\mathcal{D}) \times Seq(\mathcal{D}) \rightarrow Seq(\mathcal{D})$

Die Funktionssignaturen haben die ihrem Namen implizierte Semantik. Indexwerte beginnen bei eins, ungültige Eingabewerte liefern immer den Wert $undef$. Weitere MOF/OCL-spezifische Datentypen, wie *Bag* oder *OrderedSet*, werden bei der Formalisierung nicht unterschieden, wir verwenden stattdessen immer den Listentyp. (Praktisch tragen die anderen Sammlungsmengen außer dem Aspekt der Vernachlässigbarkeit der Reihenfolge von Elementen keine weitere Semantik.) Weiterhin

definieren wir Stapel in ähnlicher Form:

$$Stack(\mathcal{D}) =_{def} \{\text{LIFO Sequenz über } \mathcal{D}, \text{ doppelte Elemente sind erlaubt}\}$$
$$push(x, x_{Stack}) : \mathcal{D} \times Stack(\mathcal{D}) \to Stack(\mathcal{D})$$
$$pop(x_{Stack}) : Stack(\mathcal{D}) \to Stack(\mathcal{D})$$
$$top(x_{Stack}) : Stack(\mathcal{D}) \to \mathcal{D} \cup \{undef\}$$
$$size(x_{Stack}) : Stack(\mathcal{D}) \to \mathbb{N}$$
$$element(x_{Stack}, i) : Stack(\mathcal{D}) \times \mathbb{N} \to \mathcal{D} \cup \{undef\}$$

2.4.2 Metamodelle als Algebren

Algebraische Signaturen beschreiben Syntaxdefinitionen mit Sorten und darüber operierende Funktionen und haben somit Metamodellcharakter. Soll die dynamische Semantik mit ASMs oder mit vergleichbaren Kalkülen beschrieben werden, muss zunächst eine einheitliche Verbindung von MOF, dem Instanziierungskonzept aus Abschnitt 2.3.2 sowie OCL hergestellt werden. Zu diesem Zweck adaptieren und erweitern wir den Ansatz des von Richters et al. in [85] entwickelten formalen *Objektmodells* (*engl. object model*), der eine algebraische Formalisierung für UML-Klassen darstellt und im OCL-Standard Anwendung findet (vgl. nicht normativen Anhang A in [17]). Dieses Objektmodell ergänzen wir um die eingeführte logische Instanziierung und entwickeln dazu einen abstrakten ASM_{OCL}-Interpreter, der OCL und ASM zusammenführt und der eigentlichen Definition der dynamischen Semantik der MActions-Sprache dient. Diese Vorgehensweise über aufeinander aufbauende Formalisierungsschichten ist in Abb. 2.6 abgebildet.

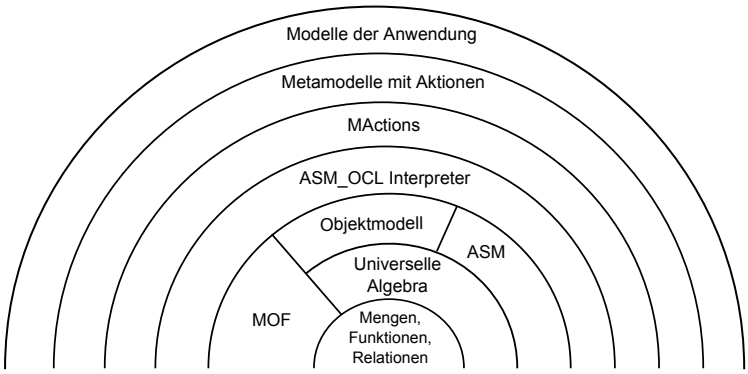


Abbildung 2.6: Formalisierungskonzept

2.4.3 Klassen- und Objektmodelle

Das algebraische Objektmodell[4] von Richters wurde zwar für UML Klassen entworfen, lässt sich jedoch durch weitere Einschränkungen auch für MOF-Metamodelle verwenden. Diese Abbildung auf Mengen, Funktionen und Relationen ist einfach gehalten und kommt ohne algebraische Konzepte höherer Stufe aus (d.h. Morphismen zwischen Algebren oder ähnlichem) und bietet sich für eine Verknüpfung mit ASM an. Wir werden diese Abbildung so konstruieren, dass sich über das in Abschnitt 2.3.2

[4]Die Namensgebung erscheint verwirrend, zumal das Objektmodell eine Denotation der *Klassen* ist — Objekte werden als konkrete Zustände dieses Modells in einer Domäne angesehen. Wir behalten hier diese Namen bei, um mit der Materie vertraute Leser nicht zu verwirren

eingeführte Instanzmodell beliebige Metaebenen erstellen und über Aktionen manipulieren lassen. Im Folgenden adaptieren wir die wichtigsten Definitionen, für weitere Details sei auf Anhang A in [17] sowie [85] verwiesen.

Die Grundidee des Objektmodells ist Metamodellklassen hinsichtlich ihrer Struktur zunächst als Bezeichner in einer Menge Class zu führen, dem Attribute, Assoziationen und Operationen zugeordnet werden. Die Semantik erhalten Elemente dieser speziellen Signatur $\mathcal{M}$ dann anschließend durch eine Algebra, die als Auswertungsfunktion I die Abbildung auf Standarddomänen für Zahlen und Mengen spezifiziert. Um die Eigenschaften des OCL- und UML-Typsystems inklusive Primitiven nachzubilden, werden Typen als getrennte Signaturen modelliert. Ein Typ $\mathcal{T}$ ist somit syntaktisch mittels einer Signatur Σ_T beschrieben, wobei die Operationsmenge Ω ihre Semantik direkt durch eine vordefinierte Auswertungsfunktion $I_{\mathcal{T}}$ bezieht. Zum Beispiel sind die primitiven Typen durch die Menge $\mathcal{T}_P = \{Boolean, Integer, Real, String\}$ beschrieben, wobei die Interpretationsfunktion diese Elemente wie folgt auf Zieldomänen abbildet:

$$I(Boolean) = \{true, false\} \cup \{\bot\} \tag{2.1}$$
$$I(Integer) = \mathbb{Z} \cup \{\bot\} \tag{2.2}$$
$$I(Real) = \mathbb{R} \cup \{\bot\} \tag{2.3}$$
$$I(String) = A^* \cup \{\bot\} \tag{2.4}$$

Als Besonderheit sei darauf hingewiesen, dass Domänen in der Regel um das Element $\bot$ für $undef$ erweitert werden, um sämtliche Abbildungen zu totalisieren. Operationen der einzelnen Typen sind erwartungsgemäß spezifiziert und im Folgenden nur in Ausnahmefällen wiederholt.

Definition 7 (Objektmodell) *Sei $\mathcal{A}$ ein Alphabet, $\mathcal{N}$ die Menge aller nichtleeren Namen über $\mathcal{A}+$. Sei weiterhin $\mathcal{T}$ die Menge aller Typen mit $\mathcal{T} = T_P \cup T_{OCL} \cup T_C \cup T_{Enum}$,[5] wobei T_C die zu den Metamodellklassen korrespondierenden Objekttypen bezeichnet, dann ist das Tuple*

$$\mathcal{M} = \langle \text{Class}, \text{Att}, \text{Op}, \text{Assoc}, \text{Meta}, associates, roles, multiplicities, \prec \rangle$$

ein Objektmodell, *mit*

I. Class $\subseteq \mathcal{N}$ *als Menge von Klassennamen („Klassifikatoren“),*

II. Att *als Menge der Operationssignaturen die jeder Klasse c seine Attribute zuordnet:* $\{\text{Att}\}_{c \in \text{Class}} = \{a : t_c \to t \mid a \in \mathcal{N}, t_c \in \mathcal{T}_c, t \in \mathcal{T}\}$

III. Op *einer entsprechende Operationszuordnung für jede Klasse:* $\{\text{Op}\}_{c \in \text{Class}} = \{\omega : t_c \times t_1 \times \cdots \times t_n \to t \mid \omega \in \mathcal{N}, t_c \in \mathcal{T}_c, t, t_1, \ldots, t_n \in \mathcal{T}\}$

IV. Assoc $\subseteq \mathcal{N}$ *als Menge aller Assoziationsnamen und den drei Funktionen:*

- *associates: eine Abbildung der Assoziation auf die teilnehmenden Klassen:*

$$associates : \begin{cases} \text{Assoc} \to \text{Class} \times \text{Class} \\ as \mapsto \langle c_1, c_2 \rangle \end{cases}$$

- *roles: eine Zuordnung der Rollennamen (Assoziationsenden) zu Assoziationen:*

$$roles : \begin{cases} \text{Assoc} \to \mathcal{N} \times \mathcal{N} \\ as \mapsto \langle r_1, r_2 \rangle \end{cases}$$

[5] T_{OCL} bezeichnet zum einen die speziellen OCL-Typen (wie z.B. *OclAny, OclVoid*) als auch Mengen- und Tupeltypen

- *multiplicities: die Zuordnung von Multiplizitäten zu Assoziationen:*

$$multiplicities : \begin{cases} \textsc{Assoc} \to M \times M \\ as \mapsto \langle M_1, M_2 \rangle, \text{ wobei } M_1, M_2 \subseteq \mathbb{N}_0 \end{cases}$$

V. $\textsc{Meta} \subset \textsc{Cl}^2$ *mit* $\textsc{Cl}^2 = \{(cl_1, cl_2) \mid cl_1 \neq cl_2, cl_1, cl_2 \in \textsc{Class}\}$ *eine partielle Ordnung über den Klassifikatoren, die die Metaobjekt Referenzierung (logische Instanziierung) nachbildet*

VI. $\prec \subset \textsc{Cl}^2$ *eine partielle Ordnung, die die Generalisierungshierarchie der Klassen nachbildet*

Es ist anzumerken, dass die obige Definition nicht nur eine Erweiterung des Objektmodells durch Meta ist, sondern in puncto Assoziationen vom Original abweicht, da MOF nur binäre Assoziationen erlaubt (vgl. auch [17], Abschnitt A.1.1.7).

Des Weiteren spezifiziert der OCL-Standard einige Zusatzbedingungen zur vorgenannten Definition, die wir hier nicht wiederholen wollen. Hauptsächlich dienen diese der Eineindeutigkeit und Konsistenz des Objektmodells, die dem Leser bereits intuitiv aus der Metamodellierung bekannt sind. Für Details s. [17], Abschnitt A.1.1.5 und A.1.1.6. Wichtiger für die weiteren Definitionen sind einige zusätzliche Hilfsfunktionen und Mengen, die im Weiteren Verwendung finden:

$$parents : \textsc{Class} \to \mathcal{P}(\textsc{Class}),\ parents(c) = \{c' | c' \in \textsc{Class} \wedge c \prec c'\} \tag{2.5}$$

$$\textsc{Att}_c^* = \textsc{Att}_c \cup \bigcup_{c' \in parents(c)} \textsc{Att}_{c'} \tag{2.6}$$

$$\textsc{Op}_c^* = \textsc{Op}_c \cup \bigcup_{c' \in parents(c)} \textsc{Op}_{c'} \tag{2.7}$$

In der Herangehensweise des OCL-Standards sind Instanzen (d.h. UML-Modelle) durch *Systemzustände* (*engl. system states*) repräsentiert. Für die weitere Betrachtung gehen wir zunächst von Modellen als statischen Instanzen aus, bevor wir bezüglich der Ausführungssemantik Modellzustände in Kapitel 4 detaillierter untersuchen. Basis der Interpretation sind Objektmengen zu jeder Klasse, bei der ein eineindeutiger Bezeichner die Existenz und Identität eines jeden Objekts bestimmt:

$$oid(\mathrm{C}) = \{\underline{obj}_1, \underline{obj}_2, \dots\} \tag{2.8}$$

Die Domäne einer Klasse $\mathrm{C} \in \textsc{Class}$ ist somit gegeben durch:

$$I_{\textsc{Class}} = \bigcup \{oid(\mathrm{C}') \mid \mathrm{C}' \in \textsc{Class} \wedge \mathrm{C}' \preceq \mathrm{C}\} \tag{2.9}$$

Folglich ist die Domäne der *Links* zwischen Objekten über das Kreuzprodukt ihrer Klassen verbindenden Assoziationen definiert:

$$I_{\textsc{Assoc}}(as) = I_{\textsc{Class}}(\mathrm{C}_1) \times I_{\textsc{Class}}(\mathrm{C}_2) \tag{2.10}$$

Zwischen T_C und Class existieren ferner zwei Funktionen, die Klassennamen auf Typen abbilden und umgekehrt:

$$typeOf : \textsc{Class} \to \mathcal{T}_\mathcal{C} \tag{2.11}$$

$$classOf : \mathcal{T}_\mathcal{C} \to \textsc{Class} \tag{2.12}$$

Jeder Systemzustand $\sigma(\mathcal{M})$ stellt eine Algebra zu einem Objektmodell $\mathcal{M}$ dar (vgl. auch [17], appendix A, Definition A.12 "System State"). Diese Algebra wollen wir im Folgenden einfach mit *Modellalgebra* bezeichnen:

Definition 8 (Modellalgebra) *Sei $\mathcal{M}$ ein Objektmodell. Dann bezeichnet die Struktur*

$$\sigma(\mathcal{M}) = \langle \sigma_{\text{CLASS}}, \sigma_{\text{ATT}}, \sigma_{\text{ASSOC}}, \sigma_{\text{META}} \rangle$$

eine Modellalgebra *(ehem. Systemzustand) des Objektmodells $\mathcal{M}$, wobei:*

- $\sigma_{\text{CLASS}}(\text{C}) \subset oid(\text{C})$ *eine endliche Menge von Objektbezeichnern ist (die Objekte)*
- σ_{ATT} *eine Menge von Funktionen die Attributen Werte zuweist:* $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(\text{C}) \to I(t)$ *für alle* $a : t_{\text{C}} \to t \in \text{ATT}^*_{\text{C}}$
- σ_{ASSOC} *eine endliche Menge von Links ist:* $as \in \text{ASSOC} : \sigma_{\text{ASSOC}}(as) \subset I_{\text{ASSOC}}(as)$*, von der wir fordern, dass sie konsistent mit der Abbildung von Multiplizitäten in multiplicity ist*
- σ_{META} *die spezielle Linkmenge der Metaobjektreferenzen ist:* $\text{META}(ref) = \sigma_{\text{META}}(ref) \subset I_{\text{CL}}(ref) \times I_{\text{CL}}(ref)$ *wobei* $I_{\text{CL}} = \{(\text{CL}_1, \text{CL}_2) \mid \text{CL}_1, \text{CL}_2 \in I_{\text{CLASS}} \vee \text{CL}_1, \text{CL}_2 \in I_{\text{ASSOC}})\}$

Bei der Namensgebung folgen wir der Konvention aus Abschnitt 2.4.1, d.h. Trägermengen werden in Großbuchstaben (z.B. SORTE) und Attribute in der „Kamelrücken-Notation“ (engl. *Camel-Back Notation*, z.B. $typeOf$) notiert. Für Trägermengen und Operationen nutzen wir weiterhin die Indexnotation:

$$\sigma_{\text{CLASS}}(\text{METACLASS}) \equiv \text{METACLASS}_{\sigma} \tag{2.13}$$

$$\sigma_{\text{ATT}}(attribute) \equiv attribute_{\sigma} \tag{2.14}$$

Ergänzend führen wir zu diesen Definitionen vereinfachte Syntaxkonventionen ein ("syntactic sugar"), die uns Zugriff auf zusammenhängende Daten der Objekte erlauben. Um den aktuellen Wert eines Attributs für ein Objekt abzufragen, nutzen wir die allgemein übliche Punktnotation auch bei Algebren, wobei der Punkt '.' als Funktion wie folgt auf Operationen zurückgeführt werden kann:

$$\underline{obj}.prop = value \equiv \sigma_{\text{ATT}}(\underline{obj})(prop) = value \text{ mit } \underline{obj} \in \sigma_{\text{CLASS}},\ value \in I(t)$$

Es ist Anzumerken, dass durch die bisherigen Definitionen die Einbettung der Semantik von OCL-Ausdrücken im Hinblick auf das orthogonale Instanzmodell offen bleibt. Bislang existiert kein OCL-Ausdruck, der die Navigation der logischen Instanziierung ermöglicht, auch wenn bereits strukturell alle Informationen in META vorgegeben sind. Zudem passt diese Abbildung noch nicht zur Repräsentation der Klasse-Objekt-Beziehung des Instanzmodells. Mit anderen Worten existieren zwei gleichwertige Sichten auf den Objektgraphen eines Metamodells:

1. Die Abbildung der Metamodellklassen nach Definition 7 und 8 auf die Mengen CLASS, dessen Typ $\mathcal{T}$ sowie σ_{CLASS} für deren Objekte. Über dieser Abbildung sind OCL-Ausdrücke definiert.
2. Das Instanzmodell aus Abb. 2.3 führt unter Anwendung der Definitionen 7 und 8 alternativ hingegen zu der generischen Objektbeschreibung OBJECT und σ_{OBJECT} für Objekte *und* Klassen.

Theoretisch ließe sich über die mathematische Modelltheorie oder ähnlichem eine Abbildung zwischen beiden Strukturen konstruieren, bei der letztlich die Signaturen und Algebren aufeinander isomorph abgebildet würden. Alternativ ließe sich eine vollständig neue OCL-Interpretationsfunktion rein über dem Instanzmodell aufsetzen, die nur auf OBJECT, SLOT etc. fußt. Beides wäre eine aufwendiges Unterfangen

und hätte im Hinblick auf die darauf aufbauende dynamische Semantik keinen Mehrwert. Deshalb verzichten wir auf eine solche vollständige Konstruktion und werden für die formale Semantik lediglich die Variante ohne Instanzmodell berücksichtigen, da durch die Erweiterung um META die selbe (genauer gesagt: eine isomorphe) Semantik angegeben werden kann.

2.5 Operationale Semantik

Operationale Semantik beschreibt Sprachsemantik als schrittweise Zustandsänderung auf einer abstrakten Maschine [26]. Seit den frühen Arbeiten zur mathematischen Formalisierung von McCarthy in den sechziger Jahren (vgl. [86]) finden heute meist Derivate der *Strukturellen Operationalen Semantik* (SOS) Anwendung, die auf Arbeiten von Gordon Plotkin zurückgeht [21][87]. Dabei ist der ursprüngliche Ansatz, den Anfangszustand in Relation zum Endzustand nach Abarbeitung eines Programms zu betrachten, einer Formalisierung von einzelnen Sprachkonzepten als schrittweise Zustandsänderung gewichen[6]. Anstatt über Regeln vollständige Ableitungsbäume für ein Programm aufzustellen, entsteht durch Rückgriff auf Transitionssysteme und Konfigurationen eine kompaktere und *direkte* Formalisierung der Ausführungssemantik einer jeden Anweisung, meist über deren abstrakte Syntax. Im Folgenden führen wir die Grundlagen und Ansätze ein, auf die auch MActions zurückgreifen und die in Kapitel 5 in Relation dazu besprochen werden.

Definition 9 (Transitionssystem) *Ein beschriftetes Transitionssystem (engl.* Labeled Transition System, LTS*) ist ein Tupel* $\mathcal{T} = (\Gamma, Act, \rightarrow)$ *mit*

1. Γ *einer Menge von Konfigurationen (Zustände),*
2. *Act eine Menge von Aktionen,*
3. $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ *der beschrifteten Transitionsrelation und*

Die Transitionsregeln werden in der Form $\gamma \xrightarrow{a} \gamma'$ *mit* $\gamma, \gamma' \in \Gamma$ *notiert, a benennt informell die ausgeführte Konfigurationsänderung*[7]*.*

Oftmals wird diese Definition variiert und um ausgezeichnete Konfigurationen erweitert (z.B. *Labeled Terminal Transition System, LTTS*), etwa:

$$\mathcal{T} = (\Gamma, Act, \rightarrow, \Gamma_0, \Gamma_\epsilon), \tag{2.15}$$

wobei $\Gamma_0, \Gamma_\epsilon \subseteq \Gamma$ Anfangs- und Endkonfiguration markieren (vgl. Abschnitt 2.7). Mittels dieser Definition lässt sich die Semantik einer Programmausführung als Sequenz $\pi = \gamma_0 \xrightarrow{a_1} \gamma_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} \gamma_n$ eindeutig beschreiben, d.h. die durchlaufenen Konfigurationen bilden einen Pfad (*engl. Trace*) durch den möglichen Zustandsraum (somit auch *Trace-Semantik*, vgl. [89], [90], [91]). Nicht terminierende Programme ergeben entweder eine unendliche Sequenz, oder sie sind *stuck at* und erreichen nie eine Endkonfiguration γ_ϵ.

Um die Transitionsregeln für eine Sprache L anzugeben, muss der Zusammenhang von Konfigurationen und abstrakter Syntax hergestellt werden. Unter der Annahme, das die abstrakte Syntax in Form von Termen über einem Alphabet oder Signatur (mit Variablen) beschrieben ist, ergibt sich für die Notation die folgende Form von Transitionsregeln (vgl. [21]):

$$\frac{\langle S_1, \sigma_1\rangle \xrightarrow{a_1} \langle S_1'\sigma_1'\rangle, \ldots, \langle S_n, \sigma_n\rangle \xrightarrow{a_n} \langle S_n', \sigma_n'\rangle}{\langle S, \sigma\rangle \xrightarrow{a} \langle S', \sigma'\rangle} \tag{2.16}$$

wobei $\langle S, \sigma\rangle \xrightarrow{a} \langle S', \sigma'\rangle$ eine Transition von einer Anweisung S mit Konfiguration σ unter einer Aktion $a \in Act$ in die Folgekonfiguration σ' und einem verbleibenden

[6] SOS wird gerne auch als *small step* Semantik bezeichnet, im Gegenzug zur 'natürlichen' oder *big step* Semantik

[7] Anmerkung: Aktionen dokumentieren hier also lediglich das geschehene und sind nicht „die treibende Kraft“ bzw. repräsentieren nicht Aktionen der modellierten Sprache, auch wenn sie in anderen Kontexten dazu verwendet werden (z.B. in CSS, s. Milner [88]).

Restprogramm S' ausdrückt. Mittels Prämissen wird ein Kriterium zur Anwendbarkeit bzw. Gültigkeit vorgegeben, das dem gleichen Schema folgt. Diese Inferenzregeln werden somit zweidimensional interpretiert:

1. Die schrittweise Abarbeitung eines Programms ist unterhalb des Transitionsstrichs notiert. Hier wird der eigentliche Effekt eines Sprachkonstrukts S bei Konfiguration σ beschrieben und es ergeben sich lineare Programmabläufe, die das Verhalten charakterisieren.
2. Die Prämissenausdrücke oberhalb des Transitionsstrichs als unmittelbare Konstituenten geben an, wann eine Regelanwendung gültig ist und erklären somit, *warum* etwas passiert. Jede Regel erhält hierüber einen vollständigen Ableitungsbaum, der im Prinzip jeden Einzelschritt einer komplexen Regel zu Zwecken der Verifikation nachweist.

Änderungen an der Konfiguration werden abstrakt über Variablen in einer semantischen Domäne in der Form $\sigma' = \sigma[m/v]$ angegeben, d.h. σ' ergibt sich durch Aktualisierung der Variablen v, welcher m als neuer Wert zugeordnet wird. Der Inferenzvorgang zeigt Züge von Termersetzungssystemen, da Regeln über Termen mit Variablen operieren und diese bei der Abarbeitung verarbeiten[8].

Während Plotkin selbst für Terme lediglich Produktionsregeln der Form $c ::= < expr_1 > | < expr_2 > | \ldots$ angibt und die Definition eines formalen Kalküls schuldig bleibt (vgl. [87]), gehen Autoren wie Bloom, Meyer, Aceto, Henessy, Barendregt et al. über algebraische Kalküle einen Schritt weiter (z.B. [93],[94],[95]). Insbesondere das oftmals verwendete *GSOS*-Regelformat ("Grand SOS") im Kontext von Prozessalgebren definiert Transitionsregeln über Terme einer Signatur Σ (vgl. Def. 1 und 3). Die abstrakte Syntax repräsentieren geschlossene Terme $x_i, y_i, t \in T_\Sigma$, die sich durch Substitution ableiten lassen (s.a. [96] [95]):

$$\frac{\bigcup_{i=1}^{l} \left\{ x_i \xrightarrow{a_{ij}} y_{ij} | 1 \leq j \leq m_l \right\} \cup \bigcup_{i=1}^{l} \left\{ x_i \overset{b_{ik}}{\nrightarrow} | 1 \leq k \leq n_l \right\}}{f(x_1, ..., x_l) \xrightarrow{c} t} \tag{2.17}$$

wobei f die ausgeführte Operation darstellt ($l \geq 0$ ist ihre Stelligkeit, $m_l, n_l \geq 0$) mit t als Zielterm. Die Prämissen $t \overset{b}{\nrightarrow}$ beschreiben negative Transitionsregeln. Im Vergleich zum Transitionssystem gemäß Def. 9 sind die Konfigurationen also „nur noch" Terme[9] und die eigentliche Transitionsrelation ist $\rightarrow \subseteq \mathcal{T}_\Sigma(Var) \times Act \times \mathcal{T}_\Sigma(Var)$ (für weitere Details sei auf [96] verwiesen).

Verallgemeinert basiert eine SOS also auf dem allgemeinen Prinzip: $\langle AS, \sigma \rangle_{\text{MACHINE}}$, wobei AS die abstrakte Syntax repräsentiert, σ eine Struktur für Konfigurationen und MACHINE einen Satz von Regeln, wie die AS verarbeitet wird. Im Zuge der algebraischen Definitionen wollen wir unter Struktureller Operationaler Semantik allgemein solche Kalküle fassen, die Terme unter Schlussregeln und Konfigurationen herleiten:

Definition 10 (Strukturelle Operationale Semantik) *Sei Σ eine Signatur, Var eine Menge von Variablen und $s_i, t_i, x \in \mathcal{T}_\Sigma(Var)$ Terme. Ein* Strukturelle Operationale Semantik *für eine Sprache L ist definiert als ein Transitionssystem*

[8] Tatsächlich zeigte Buth in [92], wie SOS Regeln als Termersetzungssystem umgeformt werden können um die Regelanwendung und somit Konfigurationen zu simulieren

[9] Die Algebra ist hier quasi die Termalgebra zur Signatur Σ als einzig zulässige Interpretation, die alle notwendigen Informationen kodiert (also dem *no junk/no confusion*-Prinzip folgend)

$\mathcal{SOS} = (\Gamma, Act, \mathcal{T}_\Sigma(Var), R_\Sigma)$ *mit* R_Σ *als Regelmenge der Form:*

$$\frac{\bigcup_{i=1}^{n} \left\{ \langle s_i, \sigma \rangle \xrightarrow{a_i} \langle t_i, \sigma_i \rangle \right\} \cup \bigcup_{i=1}^{m} \left\{ \langle s_i, \sigma_i \rangle \overset{b_i}{\not\to} \right\}}{\langle s, \sigma \rangle \xrightarrow{a} \langle t, \sigma' \rangle} \tag{2.18}$$

wobei alle $a, a_i, b_i \in Act$ *annotierte Aktionen sind, alle* $\sigma, \sigma', \sigma_i \in \Gamma$ *Elemente einer Konfigurationsmenge.*

Wir sehen GSOS-Regeln als algebraisches Redukt eines $\mathcal{SOS}$ mit $\Gamma = \{\epsilon\}$ und den in 2.17 gezeigten Einschränkungen bzgl. der Konklusion (ohne Beweis). Diese allgemeine Form der Regeln soll die Bedeutung der Konfigurationen, die zum MACTIONS-Laufzeitmodell werden, betonen (s. Abschnitt 2.6.1). Für ein $\mathcal{SOS}$-Beispiel siehe Kapitel 5.2.

2.5.1 Abstract State Machines

Während der Ansatz von GSOS-Systemen ausschließlich auf Termen beruht, verfolgt der ASM-Kalkül eine Beschreibung des operationalen Verhaltens mittels sich ändernder Algebren. ASM, alias *Evolving Algebras*, erlauben dazu dynamische Definitionen der Trägermengen und Funktionen, so dass mittels *Updates* Veränderungen an den Algebren möglich werden [97]. Wir führen die wichtigsten Definitionen ein, die für unsere Formalisierung und Diskussion wichtig sind. Für weitere Details sei auf [97] verwiesen, für ein Beispiel s.a. Kapitel 5.2.

Kernidee der ASM ist es, ein Transitionssystem als Zustandsautomat aufzusetzen, bei der die Konfigurationen Γ durch Algebren beschrieben werden und somit *abstrakte Zustände* repräsentieren. Eine Transitionsregel (Update) beschreibt, wie ein Term im Folgezustand ausgewertet wird. Ein Ablauf besteht somit aus einer Folge von sich ändernden Σ-Algebren: $\pi = \mathbb{A}_0 \xrightarrow{Update1} \mathbb{A}_1 \xrightarrow{Update2} \mathbb{A}_2 \xrightarrow{Update3} \ldots$. Gurevich führt für die ASM-Definition die Unterscheidung zwischen *static* und *dynamic* Operationsfunktionen ein, wobei erstere in jedem Zustand gleich ausgewertet werden und letztere durch Updates verändert werden können[10]. Zudem wird festgelegt, dass die Sorte BOOLEAN mit fester Interpretation auf die Werte $true, false$ Teil einer jeden Signatur eines ASM ist und generell alle Funktionen mittels des Elementes $undef$ totalisiert werden[11].

Neben den Operationen ist das Verändern der Trägermengen entscheidend. Deshalb werden die Trägermengen A_s in einem Zustand $\mathbb{A}_n$ von dem „unendlichen Elementvorrat“ – dem *Superuniversum* – unterschieden (geschrieben $|\mathbb{A}|$). Elemente entstehen deshalb nicht, sondern wechseln lediglich vom Superuniversum in die Trägermenge (dem *Universum*) oder zurück. Dazu wird im Allgemeinen die Syntax **import** v R verwendet, wobei v direkt in R gebunden wird[12].

ASM-Regeln werden in der Literatur syntaktisch teilweise mit einiger Freiheit notiert[13], wir folgen dem vermeintlichen Original von Gurevich aus [97] (s.a. [83], Kapitel 2):

[10] Die weitere Unterscheidung in *controlled*, *monitored* und *shared* wird uns erst in Kapitel 5 beschäftigen

[11] $undef$ muss deshalb zu jeder Trägermenge hinzugefügt werden

[12] Alternativ ohne Bindung **extend** DOM **with** v oder DOM := DOM $\cup \{v\}$

[13] siehe z.B. [97][24][26]

Definition 11 (ASM-Regeln) *Sei Σ eine Signatur, sowie $T_\Sigma(Var)$ und $F_\Sigma(Var)$ die Menge der Terme und Formeln. Eine* Regeldefinition *wird durch das Schlüsselwort* ***Rule*** *gekennzeichnet:*

$$\textbf{Rule}\ r(x_1, \ldots, x_n) = R, \text{mit } x_1, \ldots, x_n \in Var \text{ als Parametern} \tag{2.19}$$

R ist die eigentliche Regel und besteht syntaktisch rekursiv aus den folgenden Elementen:

1. Skip-Regel*:* **skip**
2. Update-Regel*:* $f(t_1, \ldots, t_n) := s$*, mit* $t_1, ..., t_n, s \in T_\Sigma$.
3. Block-Regel*:* $R\ S$*, mit* R, S *als Regeln*
4. If-Regel*:* **if** φ **then** R **else** S **endif** *mit* $\varphi \in F_\Sigma(Var)$ *und* R, S *Regeln*
5. Forall-Regel*:* **forall** x **with** φ **do** R **enddo** *mit* R *einer Regel,* $x \in FV(\varphi)$
6. Let-Regel*:* **let** $x = t$ **in** R *mit* $x \in Var$ *und* $t \in T_\Sigma$
7. Call-Regel*:* $r(t_1, \ldots, t_n)$*, mit* $t_1, ..., t_n \in T_\Sigma$

Syntaktisch werden komplexere Regeln mittels Makros und abgeleiteter Bezeichner weiter strukturiert (z.B. [27][26]). Dabei steht die Konvention:

$$\textsc{RuleMacro}(v_1, \ldots, v_n) \equiv \mathit{Rule}(v_1, \ldots, v_n) \tag{2.20}$$

für ein Regelmakro, bei dem *Rule* über die Variablen $v_1, \ldots, v_n$ parametrisiert wird. Abgeleitete Bezeichner der Form $\mathit{Derived}(v_1, \ldots, v_n) =_{def} f(v_1, \ldots, v_n)$ mit $f \in F_\Sigma(Var)$ definieren in gleicher Form parametrisierte Terme. Die Gesamtheit aller ASM-Regeln, Makros und dazugehörige abgeleiteten Bezeichner werden in der Menge $\mathcal{R}$ zusammengefasst.

Definition 12 (ASM) *Eine Abstract State Machine (abstrakte Zustandsmaschine) ist ein Tupel $ASM = (\Sigma, \mathcal{R}, \mathbb{A}_0, \varrho)$ bestehend aus einer Signatur Σ, einer Menge von ASM-Regeldefinitionen $\mathcal{R}$, einer Σ-Algebra $\mathbb{A}_0$ als Anfangszustand und einer ausgezeichneten Regel ϱ, der Hauptregel mit $FV(\varrho) = \emptyset$.*

Transitionsregeln erhalten ihre Semantik über Mengen von sog. "Update-Sets"[14]. Ein Update als Element dieser Menge ist ein Tripel $(f, (a_1, \ldots, a_n), b)$, das angewendet auf einen Zustand $\mathbb{A}_i$ die Interpretation der Funktion f im Folgezustand $\mathbb{A}_{i+1}$ neu zu b redefiniert, d.h. ein Update U beschreibt die Transition $f_{\mathbb{A}_i}(a_1, \ldots, a_n) \xrightarrow{U} f_{\mathbb{A}_{i+1}}(a_1, \ldots, a_n) = b$. Voraussetzung für eine Menge von Updates ist deren Widerspruchsfreiheit, die über den folgenden Kalkül definiert ist (vgl. [83], Kapitel 2):

[14] Gelegentlich wird die Notation $\Delta(R, \mathbb{S})$ genutzt, um die Änderungen einer Regel R im Zustand $\mathbb{S}$ zu beschreiben.

1. $[\![\mathbf{skip}]\!]^{\mathbb{A}}_{\beta} \rhd \emptyset$
2. $[\![f(t) := s]\!]^{\mathbb{A}}_{\beta} \rhd \{(f, a, b)\}$ wenn $a = [\![t]\!]^{\mathbb{A}}_{\beta}$ und $b = [\![s]\!]^{\mathbb{A}}_{\beta}$
3. $$\frac{[\![R]\!]^{\mathbb{A}}_{\beta} \rhd U \quad [\![S]\!]^{\mathbb{A}}_{\beta} \rhd V}{[\![R\ S]\!]^{\mathbb{A}}_{\beta} \rhd U \cup V}$$
4. $$\frac{[\![R]\!]^{\mathbb{A}}_{\beta} \rhd U}{[\![\mathbf{if}\ \varphi\ \mathbf{then}\ R\ \mathbf{else}\ S\ \mathbf{endif}]\!]^{\mathbb{A}}_{\beta} \rhd U} \quad \text{wenn } [\![\varphi]\!]^{\mathbb{A}}_{\beta} = true$$
5. $$\frac{[\![S]\!]^{\mathbb{A}}_{\beta} \rhd U}{[\![\mathbf{if}\ \varphi\ \mathbf{then}\ R\ \mathbf{else}\ S\ \mathbf{endif}]\!]^{\mathbb{A}}_{\beta} \rhd U} \quad \text{wenn } [\![\varphi]\!]^{\mathbb{A}}_{\beta} = false$$
6. $$\frac{[\![R]\!]^{\mathbb{A}}_{\beta[a/x]} \rhd U}{[\![\mathbf{let}\ x = t\ \mathbf{in}\ R]\!]^{\mathbb{A}}_{\beta} \rhd U} \quad \text{wenn } a = [\![t]\!]^{\mathbb{A}}_{\beta}$$
7. $$\frac{[\![R]\!]^{\mathbb{A}}_{\beta[a/x]} \rhd U_a, \text{ für alle } a \in I}{[\![\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ R\ \mathbf{enddo}]\!]^{\mathbb{A}}_{\beta} \rhd U} \quad \text{mit } I = \{a \in |\mathbb{A}| : [\![\varphi]\!]^{\mathbb{A}}_{\beta[a/x]} = true\}$$
8. $$\frac{[\![R]\!]^{\mathbb{A}}_{\beta[a/x]} \rhd U}{[\![r(t)]\!]^{\mathbb{A}}_{\beta} \rhd U} \quad \text{wenn } r(x) = R \text{ und } a = [\![t]\!]^{\mathbb{A}}_{\beta}$$

Die Semantik der Regeln 2. und 3. artikulieren bereits, dass mehrfache *gleichzeitige* Updates mittels Blockregeln innerhalb einer Transition erlaubt sind, solange sie widerspruchsfrei sind. Damit ergibt sich eine Ausführung eines ASM nach Def. 13:

Definition 13 (ASM-Ausführungslauf) *Sei $ASM = (\Sigma, \mathcal{R}, \mathbb{A}_0, \varrho)$ eine abstrakte Zustandsmaschine und β eine Variablenbelegungsfunktion. Ein* Ausführungslauf *(kurz: Lauf) des ASM ist eine endliche oder unendliche Sequenz von Zuständen $\mathbb{S}_0, \mathbb{S}_1, \mathbb{S}_2, \ldots$, die den folgenden Bedingungen genügt:*

1. $\mathbb{S}_0 = \mathbb{A}_0$
2. *Wenn $[\![\varrho]\!]^{\mathbb{S}_n}_{\beta}$ nicht definiert oder widersprüchlich, dann ist $\mathbb{S}_n$ der letzte Zustand der Sequenz*
3. *Falls $[\![\varrho]\!]^{\mathbb{S}_n}_{\beta}$ gültig ist, so ergibt sich $\mathbb{S}_{n+1}$ induktiv unter Anwendung der Menge von Updates U aus $\mathbb{S}_n$.*

Dieses als *ASM-Basismodell* bezeichnete Regelkalkül beschreibt also im Gegensatz zu den SOS-Systemen die Abarbeitung eines Programms (als Eingabeterm) nicht durch Termveränderung, sondern ausschließlich mittels Änderung an Algebren als Konfigurationen. So gesehen formuliert der ASM-Kalkül operationale Semantik diametral entgegengesetzt zu den GSOS-Systemen (s.a. Diskussion in Kapitel 5.2). Auch wenn wir nicht in Gänze alle verschiedenen Erweiterungen von ASM einführen können, werden wir an gegebener Stelle immer wieder Zusätze und Abweichungen vom Basismodell diskutieren und in Abschnitt 2.6.1 sogar unsere eigene Erweiterung vornehmen. Diese wird insb. einen erweiterten **forall**-Operator nutzen, der unter Berücksichtigung der entstehenden Zustände die Ausführungssemantik entweder sequentiell (Standardsemantik) oder parallel ausführt (vgl. Börger et al. [98]):

do seq
  R
  S
enddo
forall v **in** φ **do seq**
  $R(v)$
enddo
forall v **in** φ **do par**
  $R(v)$
enddo

Diese Erweiterung ist notwendig, da das Basismodell sonst alle Updates immer parallel ausführen würde, was bei der Definition von imperativen Sprachkonstrukten zu einer expliziten Sequentialisierung über z.B. extra Variablen führen würde.

2.5.2 Determinismus und Zufall

Ein Ausführungslauf beschreibt bisher immer eine (potentiell unendliche) Sequenz von Zuständen, wobei jeder Zustandsübergang *deterministisch* ist[15]. Um nichtdeterministisches Verhalten zu modellieren, existiert eine Erweiterung für ASM mittels des sog. **choose**-Konstruktors (s. [97]):

choose v **in** U **satisfying** $\varphi(v)$
  R
endchoose

wobei v eine Variable über der (möglicherweise unendlichen) Trägermenge U, R eine Regel und φ eine Formel ist[16]. Die Semantik der Regeln verändert sich dadurch so, dass R unter nichtdeterministischer Auswahl *eines* Elements aus U ein Update-Set $\Delta(R, \mathbb{S})$ auswählt, das als Transition ausgeführt wird.[17] Alternativ könnten mit einer vorher festgelegten **monitored**-Funktion zufällig Elemente geliefert werden, die in einer Regel zu Entscheidungen über **if-then-else** führen. Der **choose**-Konstruktor macht diese implizite Konvention zum expliziten Konzept in ASM.

Diese Form von modelliertem Zufall steht in engem Zusammenhang mit dem Problem der *Fairness* bei konkurrenter Ausführung von Prozessen, das sich im wesentlichen im Scheduling für konkrete Implementierungen niederschlägt (s.a. Park [99]). Fairness-Eigenschaften werden wir auf Ebene der „Meta-Semantik“ der MActions ausklammern, da mögliche Parallelität von Aktionen letztlich durch die konkrete Definition einer operationalen Semantik entschieden wird (s. Abschn. 2.5.4). Auf Ebene der Metasemantik können wir nur eine Wahrscheinlichkeitsfunktion bereitstellen, so dass *echte* Zufälligkeit und Parallelität überhaupt möglich wird, auf der Fairness-Definitionen aufbauen können (vgl. 2.5.4). Für eine Diskussion siehe Kapitel 5.2.2.

2.5.3 Parallelität und Nebenläufigkeit in Operationaler Semantik

Die Beschreibung von Parallelität und Nebenläufigkeit in Programmier- und Modellierungssprachen ist ein wesentlicher Aspekt der operationalen Semantik. Es existieren eine Reihe verschiedener Modelle für Nebenläufigkeit mit unterschiedlichen

[15] Für ASM vgl. Diskussion zu "conservative determinism" vs. "local nondeterminism" in [97]; s.a. Kapitel 5

[16] Anmerkungen: v wird auch *main existential variable* genannt, **satisfying** ist syntaktisch optional und impliziert $\varphi = \top$

[17] Man müßte eigentlich von *Indeterminismus* sprechen, da es sich bei der Auswahl um ein zufälliges Ereignis handelt. Nichtdeterminismus hingegen geht auf die Automatentheorie zurück

Schwerpunkten, die im Kern meist das Konzept von Prozessen und Kommunikation über Nachrichten beinhalten (z.B. CCS [88], Π-Kalkül [100], CSP [89], TLA [101]). Im Hinblick auf Sprachsemantik fokussieren wir auf den ASM-Kalkül sowie dessen Erweiterungen. Weitergehende Diskussionen zu SOS, CCS und CSP finden sich in Kapitel 5.

Die bisher betrachteten *rein* sequentiellen Abläufe, gegeben durch ein Transitionssystem (LTS, SOS oder ASM), waren charakterisiert durch das stete Voranschreiten von einer Konfiguration zur nächsten. Dies hat sich als Konfigurationssequenz im Trace manifestiert. Lässt man eine Ausführung von mehreren Prozessen parallel zu, ergibt sich als Konsequenz bezogen auf Aktionen und Ereignissen in den Prozessen u.U. keine globale Ordnung. Anders formuliert besteht nicht für jedes Paar von Aktionen in verschiedenen Prozessen eine *vorher/nachher* Relation im Sinne von Lamports *happend_before*-Relation (vgl. [101]). Ausführungsläufe von Prozessen können somit nur durch *partiell geordnete* Abläufe (*engl. partially ordered run*) beschrieben werden. Aus der Möglichkeit dynamisch weitere Prozesse zu erzeugen, zu beenden und zu kommunizieren ergeben sich ferner Baum-, Gitter- oder Graphstrukturen, je nach modellierter Information (vgl. [90]). Unter *Linearisierung* (Sequentialisierung) eines Prozesses verstehen wir den sequentiellen Ablauf seiner eigenen Aktionen/Ereignisse. Weitergehend beschreibt eine *gemeinsame Vergangenheit* zweier Prozesse die Schnittmenge ihrer Linearisierungen.

Für ASM existiert eine Erweiterung namens *Distributed ASM*, die über *Agenten* als parallel ablaufende Prozesse Nebenläufigkeit beschreibt (vgl. [97]). Eine endliche Menge von Agenten arbeiten individuell jeweils eine Hauptregel (Programm) ab, u.U. gestartet von einem eigenen initialen Zustand. Ein Schritt (*move*) eines Agenten ist ein 'lokales' Update des globalen Zustands:

$$Update(a, \mathbb{S}) = Update(Prog(a), View(a))$$

wobei $a \in$ AGENT die Menge der Agenten, $Prog(a)$ die Hauptregel eines Agenten und $View(a)$ eine Projektion des globalen Zustands $\mathbb{S}$ als Sicht des Agenten bezeichnet. Eine zusätzliche Funktion $Self : \rightarrow$ AGENT wird gesondert für jedes Programm interpretiert und liefert den Agenten, der das Programm gerade abarbeitet. Die Semantik der Ausführungsläufe ist über partiell geordnete Abläufe definiert, wobei partiell hier eine Halbordnung meint, die *nicht* total ist. Für weitere Details sei auf Gurevich [97] verwiesen, einen komplexeren Anwendungsfall für die Sprache SDL beschreibt Prinz in [26].

Damit es zu partiellen Ordnungen beim Ablauf kommt, bedarf es neben einem gemeinsamen Satz von *geteilten* Informationen (Zustandsgrößen, Ereignisse, ausgetauschten Nachrichten, Uhren etc.) immer auch einem Satz von *rein lokalen* Informationen, die exklusiv dem Prozess zugeordnet sind. Bei verteilten ASMs sind dies die *moves* von Agenten. Das partielle Abläufe in ihrer Semantik schwer zu fassen sind beschreibt Gurevich in [102].

Für die weitere Betrachtung in dieser Arbeit nutzen wir partielle aber *totale* Ordnungen und nicht verteilte ASMs aus folgender Motivation heraus. Nehmen wir für ASM an, zwei Agenten $a1$ und $a2$ können unterschiedliche Updates $f1 := f1'$ und $f2 := f2'$ ausführen. Diese sind disjunkt (somit konsistent) und können zur Ausführung als Schritte $m1, m2$ respektive führen. Jetzt besteht die Möglichkeit, dass $m1 < m2$, $m2 < m1$, $m2 = m1$ (gleichzeitig) oder **keine** der Optionen eintritt auf Grund der (echt) partiellen Halbordnung. Von einem übergeordneten Standpunkt aus gesehen ändert sich *während* des Ablaufs allerdings nichts: der *globale*

Zustand des Systems entwickelt sich linear entlang der Zeitachse[18], nur ist er von vornherein nicht eindeutig festgelegt. Unter der Annahme, dass Updates atomar[19] ausgeführt werden, kann man für den globalen Zustand in jedem Fall eine Reihenfolge beobachten, sagen wir $[f2 := f2']_{\mathbb{S}0} \rightarrow [f1 := f1']_{\mathbb{S}1}$, dann folgt daraus also a posteriori[20] $m2 < m1$. Somit hängen partielle Vorgänge in beiden Prozessen zwar nichtdeterministisch voneinander ab, bei einem konkreten Ablauf müssen sie sich aber in eine totale Ordnung '$\leq$' bringen lassen. Selbst ein paralleler Algorithmus, der sich in Unterprozesse unterteilt, die Änderungen an Zustandsgrößen vornehmen, führt global gesehen zu einer linearen Zustandsfolge (vgl. Parallelitätsbegriff und Applikation in [103],[104]).

Aus diesem Verständnis heraus müssen sich prinzipiell (!) also Nebenläufigkeiten als Abarbeitung auf *einer* Maschine simulieren lassen, ggf. unter Anwendung von Nichtdeterminismus und detaillierter Modellierung der Prozesswechsel, Scheduling und Kommunikation (s.a. Abschnitt 2.5.2). Umgekehrt kann die Abarbeitung einer Programmanweisung in der Semantik mit kooperierenden Prozessen beschrieben werden[21], d.h. im Allgemeinen gibt es zwischen Prozessen der Sprache und deren Definition in der operationalen Semantik keine eins zu eins Beziehung. Man könnte argumentieren, dass deshalb eine operationale Semantik keine weiteren Sprachmittel für Parallelität benötigt, allerdings möchte man paralleles Verhalten und kausale Abhängigkeiten durch hinreichende Abstraktionsmittel charakterisieren *können* (nach Bedarf), ohne jeweils von Grund auf die Eigenschaften von parallelen Prozessen explizit neu zu modellieren.[22]

Diesen Zusammenhang zwischen Lokalität der Ereignisse in Prozessen, Beobachtbarkeit und Zuständen entwickeln wir weiter zur Aktionssemantik. Die Konsequenzen daraus prägen den Ansatz zur dynamischen Analyse in Kapitel 4. Für eine Diskussion hinsichtlich Berechenbarkeit und der Thematik *„true concurrency“* s.a. Kapitel 5.2.2.

2.5.4 Aktionssemantik

Dieser Abschnitt entwickelt die Prinzipien der in Kapitel 3 definierten MActions als universelle Aktionssemantik. Universell meint dabei ein generelles, zunächst von MOF unabhängiges Konzept zur Modellausführung mittels Aktionen über Strukturen. Als Motivation und Folgerung aus den Überlegungen in Abschnitt 2.5 bis 2.5.3 und der Tatsache, dass Aktionen bislang „lediglich“ Beschriftungen an Transitionen waren, eine operationale Semantik aber eigentlich die *Änderungen* an Laufzeitgrößen/Konfigurationen spezifiziert (und weniger die dadurch stattfindenden Zustandsübergänge), soll eine neue semantische Basis geschaffen werden. Zustände werden in der Aktionssemantik deshalb als nachrangig betrachtet und kommen erst im Hinblick auf eine Analyse ins Spiel.

Definition 14 (Aktionsraum) *Sei Act eine endliche Menge von Aktionen, $\mathbb{S}$ ein Zustandsraum mit $s_0 \in \mathbb{S}$ einem ausgezeichneten Anfangszustand. Dann bezeichnet*

[18] Vorausgesetzt natürlich, wir hätten die Möglichkeit, das System aus Vogelperspektive zu beobachten und nach belieben anzuhalten bzw. zu inspizieren

[19] Vgl. [97]

[20] Im Sinne eines Zufallsexperiments

[21] Dies ist eine Motivation für das Fork/Join-Sprachmittel der MActions, vgl. 3.2.2

[22] Dies entspricht dem Wunsch nach expressiven Programmiersprachen zum Formulieren von Algorithmen, die über eine „Basismaschine“ (wie z.B. Turing-Maschine, Assembler-Sprachen, u.s.w.) hinausgehen

das Tupel $\mathbf{AR} = (\mathbb{S}, Act, \mathcal{A}, s_0)$*, einen* Aktionsraum*, mit* $\mathcal{A} \subseteq \mathcal{P}(Act)$ *der Potenzmenge über* Act*. Die Menge* $\mathcal{A}$ *umfasst alle Aktionen, deren Ausführung gleichzeitig möglich ist.*

Der Aktionsraum kann zunächst als *Möglichkeitsraum* verstanden werden und unterscheidet sich vom Transitionssystem durch die fehlende Transitionsrelation sowie der Parallelität von Aktionen. Das heißt er gibt zwar vor, welche Zustände prinzipiell existieren, beschreibt aber nicht *wie* sie in Art einer Transition zusammenhängen. Bewusst sprechen wir hier von Zustands*raum*, da wir unser Modell (zunächst) nicht einschränken wollen. Ein Zustandsraum könnte also eine Algebra sein, ein MOF-Modell oder etwas gänzlich anderes (wie z.B. ein Hilbertraum $\mathbb{R}^n$). Aktionen und Aktionsmengen manipulieren den aktuellen Zustand über dem Raum.

Die Menge $\mathcal{A}$ bestimmt über ihre Elemente den Parallelitätsgrad des Aktionsraums, geschrieben $|\mathcal{A}|_{\|}$. Zum Beispiel beschreiben einelementige Aktionsmengen $\mathcal{A} = \{\{a_1\}, ..., \{a_n\}\}$ rein sequentielle Abläufe. Bei *begrenzter* Parallelität sind alle Elemente maximal n-elementig, d.h. $|\mathcal{A}|_{\|} = n$. Anwendungsbeispiele könnten z.B. Mehrkernprozessoren sein, auf denen (bis zu) n-Aktionen gleichzeitig ausgeführt werden können. Ist die leere Menge $\emptyset \in \mathcal{A}$, existieren Zustandsübergänge *ohne* das eine Aktion ausgeführt wird, sind alle Aktionsmengen mehrelementig, so können Aktionen nur im Verbund agieren etc.

Bislang sind Aktionen reine Platzhalter und besitzen keine weitere Semantik. Insbesondere sind keine Kausalitäten oder Einschränkungen an die Elementbeziehung von Aktionen in $\mathcal{A}$ beschrieben. Dies geschieht im Zuge der Semantik von Aktionen:

Definition 15 (Semantik von Aktionen) *Sei* $\mathbf{AR} = (\mathbb{S}, Act, \mathcal{A}, s_0)$ *ein Aktionsraum. Die Semantik der Aktionen ist eine Abbildung* $\delta : \mathbb{S} \times \mathcal{A} \to \mathbb{S}$ *von der wir fordern:*

1. $\delta(s, a_1) = \delta(s, a_2) \Rightarrow a_1 = a_2$ *(Eindeutigkeit von Aktionen)*
2. $\delta(s, a)$ *ist für alle* $a \in Act$ *definiert (alle Aktionen erhalten eine Semantik)*

Von den Aktionen fordern wir insbesondere keine Assoziativität. Da Aktionen im Allgemeinen nicht in jedem Zustand Anwendung finden können, beschreibt δ in der Regel eine partielle Abbildung, ähnlich zur Transitionsrelation. Im Gegensatz zum Transitionssystem kann die Semantik einer Aktion also z.B. im Aktionsraum $\mathbb{R}$ auch eine reelle Funktion oder numerische Berechnungsvorschrift sein, bei Objektgraphen eine Modelltransformation etc. Die Reihenfolge von Aktionen wird durch Prozesse vorgegeben:

Definition 16 (Prozess) *Sei* $\mathbf{AR}$ *ein Aktionsraum,* δ *die Semantik von Aktionen des Raums. Ein Prozess* $P = (Act_P, \mathbf{AR}, \delta)$ *ist eine unabhängige Entität, die durch eine Aktionsmenge* $Act_P \subseteq Act$*, dem Verhalten des Prozesses, charakterisiert ist und diese ausführt.*

Somit bestimmen Prozesse die Aktionsausführung aktiv und beschreiben zusammen mit der Transitionsrelation δ implizit die Kausalität zwischen Aktionen, d.h. Reihenfolgen werden über Prozesse festgelegt, ob eine Aktion ausgeführt werden kann (und wie) beschreibt δ. Ein Prozess muss seine Aktionen nicht sequentiell ausführen, sondern kann nach eigenen Vorschriften/Kontrollstrukturen funktionieren. Zum Beispiel könnte ein Prozess zyklisch Aktionen ausführen oder Aktionen bedingt von einer Entscheidung über Wertänderungen im Zustandsraum auslösen.

Mehrere Prozesse können gleichzeitig im selben Aktionsraum ausgeführt werden, wenn sie dieselbe Aktionssemantik δ teilen. Das gleichzeitige Ausführen von

Aktionen ist dann durch das Element $\{a_{P_1}, ..., a_{P_n}\} \in \mathcal{A}$ beschrieben und muss selbstredend im Aktionsraum möglich sein. So gesehen „feuern“ Prozesse ihre Aktionen aktiv und der Aktionsraum ist eine Art „Schiedsrichter“, ob die Kombination zulässig ist. Sollte sie zulässig sein, wird die Transition anhand von δ ausgeführt. Falls nicht, geschieht keine Änderung. Fairness zwischen Prozessen ist gegeben als gleich verteilte Wahrscheinlichkeit für die Auswahl der Prozesse, die zu einem Zeitpunkt ausgeführt werden. Prozesse können durch Aktionen initiiert werden, d.h. ausgewählte Aktionen können die Menge P aller Prozesse erweitern.

Definition 17 (Ausführungslauf) *Sei* $\mathbf{AR}$ *ein Aktionsraum,* δ *die Semantik von Aktionen des Raums und* $P_1, ..., P_n$ *Prozesse der Form* $P = (Act_P, \mathbf{AR}, \delta)$*. Ein Ausführungslauf ist beschrieben als sequentielle Ausführung von Aktionsmengen mit* $t = 0, 1, 2, ...$ *der Form*

$$A_t = \bigcup_{i=1..n} A_{P_i,t} \subseteq Act_P,$$

wobei $A_{P_i,t}$ *die Aktionen zu einem Zeitpunkt* i *bezeichnen. Diese Aktionsfolge kann als Folge* π *zusammen mit den jeweils erreichten Zuständen aufgezeichnet werden.*

Als Resultat eines Ausführungslaufs erhalten wir einen Trace, der über Zustände und Aktions*mengen* beschrieben ist (als geordnete Multimenge):

$$\pi = \{\{s_0 \xrightarrow{\mathcal{A}_1} s_1 \xrightarrow{\mathcal{A}_2} s_2 \ldots \xrightarrow{\mathcal{A}_n} s_n | s_i \in \mathbb{S}, \mathcal{A}_i \in \mathcal{A}\}\} \tag{2.21}$$

Für Elemente des Traces nutzen wir die Notationen $(s_i, \mathcal{A}_i, s_{i+1}) \equiv s_i \xrightarrow{\mathcal{A}_i} s_{i+1}$ synonym. Die Stelle des Auftreten von $\mathcal{A}_i$ im Trace wird mit $o(\mathcal{A}_i)$ angegeben. Die strikte Ordnung der Elemente im Trace ist durch $(s_i, \mathcal{A}_i, s_{i+1}) < (s_j, \mathcal{A}_j, s_{j+1})$ beschrieben, wenn $o(\mathcal{A}_i) < o(\mathcal{A}_j)$. Ein *Segment* $S(i, j) \subseteq \pi$ ist eine Untermenge, die alle Elemente zwischen $o(\mathcal{A}_i)$ und $o(\mathcal{A}_j)$ enthält.

Aus der Ordnung der Einträge im Trace folgt eine lineare Ordnung über die ausgeführten Aktionen im Aktionsraum:

Definition 18 (Reihenfolge von Aktionen) *Die Relation* $\leq$*:* $Act \times Act$ *über einem Trace* π *mit* $a_1 \in \mathcal{A}_1, a_2 \in \mathcal{A}_2$ *ist eine lineare Ordnung und wie folgt definiert:*

1. $a_1 = a_2$ *gdw.* $\mathcal{A}_1 = \mathcal{A}_2$*, d.h.* $a_1, a_2 \in \mathcal{A}$
2. $a_1 < a_2$ *gdw.* $(s_i, \mathcal{A}_1, s_{i+1}) < (s_j, \mathcal{A}_2, s_{j+1})$

Aus der Definition ist ersichtlich, dass '$\leq$' reflexiv, transitiv, antisymmetrisch und total ist, wobei insbesondere die letzte Eigenschaft für weitere Analysen wichtig ist. Als besondere Form der Prozesse verstehen wir *Threads*[23], die jeweils genau eine Aktion ausführen:

Definition 19 (Thread) *Sei* P *ein Prozess* $P = (Act_P, \mathbf{AR}, \delta)$*. Ein Prozess ist ein* Thread*, wenn bei jedem Ausführungsschritt nur ein* $a \in Act_P$ *ausgeführt wird.*

Es sei angemerkt, dass unser Modell von Aktionen und Prozessen keine Synchronisation oder Kommunikation enthält. Diese kann indirekt durch Aktionen und Strukturen modelliert werden, je nach gewünschtem Parallelitätsverhalten (siehe z.B. Appendix A). Weiterhin existieren keine Semaphoren oder ähnliches für exklusiven

[23]Das Deutsche Word „Faden“ spiegelt die Bedeutung gut wider, wird allerdings aufgrund der Verständlichkeit nicht verwendet.

Zugriff auf „Resourcen“, da auftretende Konflikte der Aktionen nicht zu einem Anhalten der Maschine führen, sondern zum erneuten Versuch einer Ausführung (quasi „aktives Warten“ der Prozesse).[24] In diesen Punkten unterscheidet sich das semantische Basismodell z.B. grundlegend von ASM (vgl. 2.5.1).

Die in diesem Abschnitt eingeführten Konzepte dienen als Basis für die Definition der MActions in Kapitel 3.1. Dort werden wir dem Prinzip nach als Zustandsraum sämtlich mögliche Instanzen eines MOF-Laufzeitmetamodells als Strukturen einsetzen, und einen elementaren Satz von Aktionen zum Erzeugen und Manipulieren dieser Modelle.

2.6 Interpreter für die Aktionssemantik

Dieser Abschnitt legt das Fundament für eine formale Definition der MActions-Aktionssemantik als operationales Kalkül für εMOF. Konzeptionell werden MActions meta-zirkulär über sich selbst beschrieben, allerdings geben wir eine alternative Formalisierung mittels eines ASM-Interpreters an, um mathematisch formal die Semantik im Hinblick auf eine Modellanalyse festzulegen.

In Abschnitt 2.4 wurden Metamodelle allgemein auf das Objektmodell und Systemzustände statisch abgebildet. Mit dem ASM-Kalkül könnte man über dynamische Definitionen der Trägermengen und Funktionen bereits operational Objektgraphen verändern und Verhalten beschreiben (vgl. Abschnitt 2.5.1). Voraussetzung ist ein Satz von Regeln, der die Algebren metamodellkonform verändert. Dabei entsprechen diese „beliebigen“ Updates allerdings nicht der Aktionssemantik. Zudem fehlt die Interpretation von OCL-Ausdrücken. Deshalb soll eine Zusammenführung der algebraischen Repräsentation von MOF und OCL das gewünschte Fundament liefern.

2.6.1 ASM_{OCL}: OCL-Interpreter über ASM

Das Objektmodell aus Abschnitt 2.4.3 liefert eine Denotation von OCL-Ausdrücken als algebraisches Objektmodell, die im Kontext von ASMs verwendet werden kann. Da die Interpretationsfunktion von OCL Ausdrücke als Terme in eine Algebra auswertet, bleibt zu klären, wann und wie ein Ausdruck im Zuge der dynamischen Updates ausgewertet wird. Das heißt die „operationale Komponente“ in ASM muss in geeigneter Form mit der Interpretationsfunktion $I_{OCL}[\![expr]\!]$ integriert werden. Das Ziel wäre eine Verwendung von OCL etwa in Regeln wie:

$$\textsc{Var} := self.value- > collect(x) \tag{2.22}$$

wobei diese Zuweisung einer OCL-Auswertung an die dynamische Domäne Var kompatibel mit der Klassenabbildung des jeweiligen Metamodells sein muss. Um dies zu erreichen ist neben der OCL-Signatur auch ein Eingriff in die Interferenzregeln und dem Evaluierungsmechanismus von ASM nötig, um die Termauswertung um I_{OCL} zu erweitern und das Verhalten von Updates (besonders für Variablenverwendung) und OCL zusammenzuführen. Basierend auf dem Objektmodell und einer Modellalgebra ist die Semantik eines OCL-Ausdrucks $expr \in Expr_t$ vom Typ t in einer Umgebung Env wie folgt beschrieben:

$$I_{OCL}[\![expr]\!] : Env \rightarrow I_{OCL}(t) \tag{2.23}$$

[24] Durch die zufällige Auswahl der zu einem Zeitpunkt auszuführenden Prozesse ist eine Fortführung prinzipiell garantiert, so fern nicht Vorbedingungen der Aktionen oder die Kontrolllogik eines Prozesses dies unterbindet.

d.h. I_{OCL} ordnet jedem Ausdruck einen Wert in der Domäne seines Typs zu. Die Umgebung $Env = (\sigma, \beta)$ besteht aus einer Modellalgebra σ und einer Variablenbelegung β, die alle freien Variablen einen Wert ihres Typs zuweist: $\beta : Var_t \to I_{OCL}(t)$. Als leichtgewichtige Alternative könnte man eine neue Operation *Eval* einführen, die das Problem isoliert löst:

$$\textsc{Var} := Eval(expr) \tag{2.24}$$

Um diese rein funktionale Definition durch *Eval* in ASM einzubetten, müsste mittels geeigneter Trägermengen die OCL-Ausdruckssemantik erhalten bleiben und insbesondere Variablen vor der Auswertung mit aktuellen Werten des abstrakten Zustands belegt werden. Da die Semantik der Update-Regeln gemäß Def. 12 ff. nicht erweitert werden müsste, sondern „nur“ die Termauswertung, verfolgen wir den zweiten Lösungsweg. Dazu legen wir ASM einen erweiterten Interpretationsmorphismus Υ zu Grunde, der die normale Interpretation für ASM-Terme nutzt und für *Eval* die OCL-Interpretation I_{OCL} ruft. Des Weiteren schränken wir die Nutzung von *Eval* auf die rechte Seite der Update-Regeln ein, d.h.:

$$[\![f(t) := s]\!]^{\mathbb{A}}_{\beta} \rhd \{(f, a, b)\} \quad \text{wenn } a = [\![t]\!]^{\mathbb{A}}_{\beta} \text{ und } b = [\![s]\!]^{\mathbb{A}}_{\beta} \text{ und } f \neq Eval \tag{2.25}$$

Somit kann die Interpretation als vollständig gekapselte „Teilmaschine“ transparent innerhalb eines ASMs ablaufen, da der OCL-Ausdruck den abstrakten Zustand ja nicht verändert. Genauer gesagt verbindet *Eval* den OCL-Ausdruck durch Interpretation in einer eigenen Umgebung Env und liefert als Resultat eine Menge von Elementen aus der Ziel-Domäne der Zuweisung. Damit die Kompatibilität hergestellt werden kann, ergibt sich die Notwendigkeit eines gemeinsamen Satzes von Domänen:

Definition 20 (Laufzeitmodell ASM_{OCL}-Interpreter) *Sei $\mathcal{M}_{\mathcal{A}}$ das Objektmodell für ein Metamodell $\mathcal{A}$, dann bezeichnet $\mathcal{M}_{\text{ASMOCL}} \supseteq \mathcal{M}_{\mathcal{A}}$ das* Laufzeitmodell *eines ASM_{OCL}-Interpreter für $\mathcal{A}$, notiert als $\mathcal{M}_{\text{ASMOCL}}(\mathcal{A})$, mit den folgenden Sorten und Operationen*

I. $\textsc{Class}_{\text{ASMOCL}} = \{\text{MThread}, \text{MContext}, \text{MIncarnation}, \text{PlaceToInc}, \text{ThreadState}, \text{OclAny}, \text{OclInvalid}, \text{String}, \text{Integer}, \text{Natural}, \text{Real}, \text{Boolean}, \text{EnumValue}, \text{Ocl}\} \cup \textsc{Class}_A \cup \textsc{Class}_{\text{OclTypes}}$

II. $\textsc{Att}_{\text{ASMOCL}}$ *enthält neben* $\textsc{Att}_A$ *zusätzlich die folgenden Attributabbildungen:*

- $name : \text{MThread} \to \text{String}$
- $objects : \text{MIncarnation} \to \text{OclAny}$
- $self : \text{MContext} \to \text{OclAny}$

III. $\textsc{Assoc}_{\text{ASMOCL}} = \{Stack, CurrentPosition, StackValues, Incarnations\} \cup \textsc{Assoc}_A$ *mit den Eigenschaften*

- $associates = \{Stack \mapsto \langle \text{MThread}, \text{MContext}\rangle, Incarnations \mapsto \langle \text{MContext}, \text{PlaceToInc}\rangle, StackValues \mapsto \langle \text{PlaceToInc}, \text{MIncarnation}\rangle\}$
- $roles = \{Stack \mapsto \langle thread, stackFrames\rangle, Incarnations \mapsto \langle context, incarnations\rangle, StackValues \mapsto \langle incarnations, value\rangle\}$
- $multiplicities = \{Stack \mapsto \langle \{1\}, \mathbb{N}_0\rangle, Incarnations \mapsto \langle \mathbb{N}_0, \mathbb{N}_0\rangle, StackValues \mapsto \langle \mathbb{N}_0, \{0, 1\}\rangle\}$

IV. $\textsc{Meta}_{\text{ASMOCL}} = \textsc{Meta}_A$
VII. $\prec_{\text{ASMOCL}} = \{(C, \textsc{OclAny}) \mid \forall C \in \textsc{Class}_A . \neg\exists S \in \textsc{Class}_A .\ \prec_A (C, S))\}$
VIII. Darüber hinaus existieren noch weitere Operationen:

1. *Funktionsfamilie* $Eval_T : \textsc{Ocl} \times \textsc{Env} \to T$,
   wobei $T \in \{\textsc{OclAny}, Seq(\textsc{OclAny}), \textsc{String}, \textsc{Boolean}, \textsc{Integer}, \textsc{Natural}, \textsc{Real}, \textsc{EnumValue}\}$
2. $OclInvalid :\to \textsc{OclInvalid}$
3. *Alle OCL-Operationen für die Sammlungsmengen* $\textsc{Sequence}, \textsc{OrderedSet}, \textsc{Set}, \textsc{Bag}$ *sowie den Primitive Typen*

Wie aus der Definition ersichtlich sind allgemeine Laufzeitstrukturen für Threads durch MThread, Stapel- und Variablenkontexte durch MContext und MIncarnations mit dem Metamodell und OCL verbunden. Die OCL Metamodellabhängigkeiten schränken wir auf *Essential OCL* ein, dass nur vom EMOF-Paket abhängt, da es explizit die folgenden Klassen referenziert: Property, Operation, Parameter, TypedElement, Type, Class, DataType, Enumeration, PrimitiveType und EnumerationLiteral (vgl. 13.1. und 13.2. in [17]). Neben den OCL-Domänen, wie z.B. OclAny, bildet die Funktionsfamilie *Eval* hauptsächlich die Integration für die Auswertung von OCL-Ausdrücken ab, welche als Menge in Ocl enthalten sind. Die Parametrisierung der Auswertungsumgebung erfolgt dabei über eine Umgebung *Env*, wie oben beschrieben. Bewusst sind diese Modellsignaturen so gewählt, dass sie dem Laufzeitmodell der in Kapitel 3, Abschnitt 3.2.5 definierten MActions entsprechen und möglichst eins zu eins die dortigen Klassen abbilden. Mit anderen Worten leitet sich die Definition von $\mathcal{M}_{\text{ASMOCL}}$ unter Anwendung der Def. 7 nahezu vollständig aus den Klassen des Laufzeitmodells und der OCL-Syntaxdefinition ab.

Um dieser Signatur das gewünschte Leben einzuhauchen, müssen die Trägermengen und Funktionen mittels einer Algebra $\sigma(\mathcal{M}_{\text{ASMOCL}})$ festgelegt und die Auswertungsfunktionen, wie eingangs erwähnt, zusammengefügt werden. Die daraus resultierende dynamische Modellalgebra ASM_{OCL} soll einen *Multi-Agenten-ASM* mit totaler Laufzeitordnung der Aktionen simulieren, bei dem OCL-Ausdrücke genutzt werden können, um über Objektgraphen zu navigieren. Aufgrund der echt partiellen Ordnung von Verteilten ASMs werden wir dazu mehrere Prozesse (bzw. Threads) parallel durch das ASM-Basismodell als Kern der Laufzeitsemantik der MActions simulieren (s.a. 2.5.3).

Die Signatur für Threads verhält sich ähnlich der der ASM-Agenten und ermöglicht Zugriff auf Laufzeiteigenschaften wie z.B. Hinzufügen von Threads, Abfragen des Laufzustands etc. [25] Besonderes Augenmerk gilt der Definition von *Eval*:

Definition 21 (*ASM_{OCL}*-Interpreter) *Die abstrakte Maschine* $ASM_{OCL} = \langle \Sigma, \mathcal{T}, \Theta \rangle$ *ist ein spezieller Multi-Agenten ASM mit Signatur* $\Sigma \supseteq \mathcal{M}_{ASMOCL}$*, Aktualisierungsregeln* $\mathcal{T}$ *über den Domänen* Θ*, wobei* $\Theta \supseteq \sigma(\mathcal{M}_{\text{ASMOCL}})$*. Dabei leiten sich die* σ *Trägermengen regulär nach Def. 8 als dynamische ASM-Domänen ab, mit den folgenden Besonderheiten:*

I. $\textsc{MThread}_\sigma$ *bildet Zugriff auf die Menge abstrakt parallellaufender Entitäten, die wie alle Domänen dynamisch erweitert werden kann.*
II. Neben der Objektstruktur werden Primitive durch die elementaren Mengen der OCL-Semantik definiert:

[25] Zur Diskussion und Vergleich von ASM als den „äußeren, unveränderbaren“ Rahmen der Semantik und dessen Bezug zum Theorem der Universellen Transformationsmaschine aus Kapitel 3.1.2 s. 5.2

1. $\text{BOOLEAN}_\sigma = \{true, false\} \cup \{\bot\}$
2. $\text{INTEGER}_\sigma = \mathbb{Z} \cup \{\bot\}$
3. $\text{NATURAL}_\sigma = \mathbb{N}_0 \cup \{\bot\}$
4. $\text{REAL}_\sigma = \mathbb{R} \cup \{\bot\}$
5. $\text{STRING}_\sigma = A^* \cup \{\bot\}$
6. $\text{ENUMVALUE}_\sigma = A^* \cup \{\bot\}$
7. *Gleiches gilt für die OCL-Sammlungsmengen und deren Operationen (also z.B.* $\text{SEQ}_\sigma, \text{ORDEREDSET}_\sigma, \text{SET}_\sigma, \text{BAG}_\sigma$ *etc.)*

III. OCL_σ *ist die Menge von OCL-Ausdrücken, die mit Eval ausgewertet werden können:* $Eval_t : \text{OCL}_\sigma \times \text{ENV} \to I^*_{OCL}(t)$*, mit* $Eval_t(expr, \varepsilon) = I^*_{OCL}(expr_t)$*, wobei* I^*_{OCL} *die erweiterte Auswertung in einer Umgebung* $\varepsilon = (\sigma(\mathcal{M}_{\text{ASMOCL}}), \beta^*)$ *stattfindet, bei der* $\beta^* : Var_t \to I(t)$ *die Variablenmenge* Var_t *jeweils aus dem aktuellen Zustand des ASM bereitstellt.*

2.6.2 Kernsemantik der MActions

Die abstrakte ASM_{OCL}-Maschine hat bislang alle Freiheitsgrade, insbesondere was Aktualisierungsregeln, zusätzliche Domänen und Parallelitätseigenschaften angeht. In diesem Abschnitt sollen die Konzepte soweit eingeschränkt werden, dass im Vorgriff auf Kapitel 3 der Kern der MACTIONS-Aktionssemantik in puncto Abarbeitung, Reihenfolge und Parallelität von Aktionen bereits vorgegeben ist. Anschließend können dann die verschiedenen Arten von Aktionen durch einzelne Regeln ergänzt werden.

Zu diesem Zweck muss zunächst der Zugriff auf ein *Aktionsmodell* beschrieben werden können. Das heißt Regeln müssen auf ein „MACTIONS-Programm“ lesend zugreifen können, um diese zu interpretieren. Eine einzelne Aktion beschreibt eine elementare Veränderung des Laufzeitmodells. Aktionen bilden im Verbund ein Netz, das Knoten für Knoten sequentiell abgearbeitet wird. Mehrere dieser Netze können durch Threads parallel ausgeführt werden und durch Aktivitäten und Operationen modularisiert werden. Um diese Netzstruktur als abstrakte Syntax algebraisch zu fassen, bedienen wir uns erneut der Abbildung über ein Objektmodell (vgl. Abschnitt 2.4.2):

Definition 22 (Aktionsmodell) *Sei* $\mathcal{M}_{\text{ASMOCL}}(\mathcal{A})$ *ein Laufzeitmodell für ein Metamodell* $\mathcal{A}$*, dann besteht ein* Aktionsmodell *für* $\mathcal{A}$ *als Objektmodell zusätzlich aus den folgenden Elementen:*

I. $\text{CLASS}_{\text{MACTIONS}} = \{$ENAMEDELEMENT, MACTION, MACTIVITY, MACTIVITYEDGE, MACTIVITYNODE, MACTIVITYPARAMETER, MBEHAVIOUR, MOPERATION, MCLASS, MCONTROLFLOW, MCONTROLNODE, MINITIALNODE, MDECISIONMERGENODE, MFINALNODE, MFORKJOINNODE, MCREATEACTION, MINVOCATIONACTION, MITERATEACTION, MPLACE, MOBJECTNODE, MPIN, MASSIGNACTION, MQUERYACTION, MINPUT, MOUTPUT, MACTIONGROUP, MATOMICGROUP$\}$

II. $\text{ATT}_{\text{MACTIONS}} = \{$*name* : ENAMEDELEMENT $\to$ STRING, *guardExpression* : MACTIVITYEDGE $\to$ OCL, *assignExpression* : MASSIGNACTION $\to$ OCL, *expression* : MDECISIONMERGENODE $\to$ OCL, *terminateThread* : MFINALNODE $\to$ BOOLEAN,

$startThread$: MInvocationAction → Boolean,
$iterateExpression$: MIterateAction → Ocl,
$constant$: MObjectNode → Boolean,
$queryExpression$: MQueryAction → Ocl,
$tracePoint$: MActivityNode → Boolean}

III. $\text{Assoc}_{\text{MActions}} = \{IncomingEdges, OutgoingEdges, OwnedNodes, OwnedEdges, OwnedPins, OwnedParameters, OwnedMembers, RepresentedPlaces, RedefinedBehaviour, InvokedBehaviour\}$ mit den Eigenschaften

- $associates = \{CurrentNode \mapsto \langle$MContext, MActivityNode$\rangle$,
  $IncomingEdges \mapsto \langle$MActivityEdge, MActivityNode$\rangle$,
  $OutgoingEdges \mapsto \langle$MActivityEdge, MActivityNode$\rangle$,
  $OwnedNodes \mapsto \langle$MBehaviour, MActivityNode$\rangle$,
  $OwnedEdges \mapsto \langle$MActivityNode, MActivityEdge$\rangle$,
  $OwnedPins \mapsto \langle$MAction, MPin$\rangle$,
  $OwnedParameters \mapsto \langle$MBehaviour, MActivitiyParameter$\rangle$,
  $OwnedMembers \mapsto \langle$MActionGroup, MActivityNode$\rangle$,
  $RepresentedPlaces \mapsto \langle$MActivityParameter, MPlace$\rangle$,
  $RedefinedBehaviour \mapsto \langle$MBehaviour, MBehaviour$\rangle$,
  $InvokedBehaviour \mapsto \langle$MInvocationAction, MBehaviour$\rangle\}$
- $roles = \{CurrentNode \mapsto \langle action, currentNode\rangle$,
  $IncomingEdges \mapsto \langle incoming, source\rangle$,
  $OutgoingEdges \mapsto \langle outgoing, target\rangle$,
  $OwnedNodes \mapsto \langle owningBehaviour, ownedNodes\rangle$,
  $OwnedEgdes \mapsto \langle owningNode, ownedEdges\rangle$,
  $OwnedPins \mapsto \langle action, ownedPins\rangle$,
  $OwnedParameters \mapsto \langle behaviour, parameters\rangle$,
  $OwnedMembers \mapsto \langle group, memberNodes\rangle$,
  $RepresentedPlaces \mapsto \langle parameter, places\rangle$,
  $RedefinedBehaviour \mapsto \langle redefiningBehaviour, redefinedBehaviour\rangle$,
  $InvokedBehaviour \mapsto \langle invokingBehaviour, invokedBehaviour\rangle\}$
- $multiplicities = \{CurrentNode \mapsto \langle\{1\}, \{1\}\rangle$,
  $IncomingEdges \mapsto \langle\mathbb{N}_0, \{1\}\rangle$,
  $OutgoingEdges \mapsto \langle\mathbb{N}_0, \{1\}\rangle$,
  $OwnedNodes \mapsto \langle\{1\}, \mathbb{N}_0\rangle$,
  $OwnedEdges \mapsto \langle\{1\}, \mathbb{N}_0\rangle$,
  $OwnedPins \mapsto \langle\{1\}, \mathbb{N}_0\rangle$,
  $OwnedParameters \mapsto \langle\{1\}, \mathbb{N}_0\rangle$,
  $OwnedMembers \mapsto \langle\{0, 1\}, \mathbb{N}_0\rangle$,
  $RepresentedPlaces \mapsto \langle\{0, 1\}, \mathbb{N}_0\rangle$,
  $RedefinedBehaviour \mapsto \langle\{1\}, \{1\}\rangle$,
  $InvokedBehaviour \mapsto \langle\{1\}, \{1\}\rangle\}$

IV. $\text{Meta}_{\text{MActions}} = \{\}$

VII. $\prec_{\text{MActions}} = \{$(MActivityNode, ENamedElement),
(MActivityEdges, ENamedElement),
(MAction, MActivityNode), (MActivity, EClass),
(MActivity, MBehaviour), (MActivityParameter, EParameter),
(MAssignAction, MAction), (MControlFlow, MActivityEdge),
(MControlNode, MActivityNode), (MCreateAction, MAction),
(MDecisionMergeNode, MControlNode), (MFinalNode, MControlNode),

(MForkJoinNode, MControlNode), (MInitialNode, MControlNode),
(MInvocationAction, MAction), (MIterateAction, MActionGroup),
(MObjectNode, MPlace), (MOperation, EOperation),
(MOperation, MBehaviour), (MPin, MPlace),
(MPlace, ETypedElement), (MPlace, MActivityNode),
(MQueryAction, MAction), (MInput, MAction),
(MOutput, MAction), (MActionGroup, MAction),
(MAtomicGroup, MActionGroup)}

Es sollte nicht überraschen, dass sich dieses Objektmodell aus dem abstrakten Syntax-Metamodell der MActions ableitet, welches in Kapitel 3 detailliert erklärt wird. (s. Abschnitt 3.2 und 3.3). Die zugehörige Modellalgebra $\sigma(\mathcal{M}_{\text{MActions}})$ bezeichnet das *algebraische Laufzeitmodell* der MActions. Da wir im Folgenden immer von „der einen“ Algebra ausgehen (d.h. dem Objektmodell), werden wir die Indizes vernachlässigen und die für ASM gängigen Notationen verwenden.

Die Grundidee des strukturellen Aufbaus von Aktionen realisiert MActivityNode und MActivityEdge. Jeder Ablauf soll von Knoten zu Knoten voranschreiten. Dabei existieren neben Aktionsknoten (Ableitungshierarchie MAction $\prec$ MActivityNode) auch Kontrollfluss- und Datenflussknoten (Ableitungshierarchie MControlNode $\prec$ MActivityNode), die bei der Ausführung berücksichtigt werden. Pro Thread können neben der ausgehenden Transition für den Kontrollfluss immer weitere Transitionen für den Datenfluss existieren. Das Voranschreiten des Kontrollflusses regelt $NextNode$(MThread). Diese Regel ist so definiert, dass auch Aktionsgruppen mit inneren Aktionen berücksichtigt werden. Für weitere Navigation im abstrakten Syntaxmodell existieren die Funktionen:

$entryPoint :\rightarrow$ MActivity
$InitialNode$: MActivity $\rightarrow$ MActivityNode
$InputPlaces$: MActivityNode $\rightarrow Seq$(MPlace)
$OutputPlaces$: MActivityNode $\rightarrow Seq$(MPlace)
$ConnectedInnerPlaces$: MIterateAction $\rightarrow Seq$(MPlace)
$ToSibling$: MActivityEdge $\rightarrow$ MActivityNode
$ToInnerNode$: MActivityEdge $\rightarrow$ MActivityNode
$Parent$: MActivityNode $\rightarrow$ MActivityNode

Reflexion über den gerade zu verarbeitenden Knoten liefert die Funktion **is**, die zur Fallunterscheidung für die meisten ASM-Regeln dient:

$$\textbf{is} : \text{Class} \rightarrow \text{Boolean mit } x \textbf{ is } T := \begin{cases} true & x \in \sigma_T \\ false & x \notin \sigma_T \end{cases}$$

Das jeweils abzuarbeitende Verhalten kann über die Funktion *ResolveBehaviour* abgerufen werden, welche mittels des aktuellen Kontextes eine mögliche `self`-Belegung an einem Pin der Aufrufaktion auswertet:

$ResolveBehaviour$: MContext $\times$ MInvocationAction $\rightarrow$ MActivity

Für MOperations wird unter Berücksichtigung von überschriebenen Operationen polymorph das Verhalten geliefert, was dem Objekttyp am `self`-Pin zur Laufzeit

entspricht. Des Weiteren werden Objekte mittels folgender Methoden erzeugt, gefunden und manipuliert:

$GenOid :\rightarrow$ Class
$Classifier$: MPlace $\rightarrow$ Class
$CurrentObjects$: MContext $\times$ MPlace $\rightarrow Seq($OclAny$)$
$AttToChange$: MAssignAction $\rightarrow$ Att$_{\text{A}}$

Neue Objekte entstehen über *GenOid*, die als Domäne die Objektmenge *oid*(Class) für jede Klasse getrennt definiert. Initial sind die Objektmengen *oid*(Class) für alle Klassen des Laufzeitmetamodells leer. Der Typ eines Platzes kann mittels *Classifier* bestimmt und über *CurrentObjects* die im aktuellen Stapelkontext gebundenen Objekte erfragt werden. Es wird immer eine Sequenz von Objekten geliefert, sowohl für Plätze mit und ohne Multiplizität (d.h. `[0..1]`, `[1..1]`). Für Änderungen von Attributwerten mittels MAssignActions liefert *AttToChange* die notwendige Navigation durchs Metamodell.

An den Stellen, an denen OCL-Abfragen ausgewertet werden, kommt die Funktionsfamilie *Eval* zum Einsatz:

$Eval : Expr_t \times$ Env $\rightarrow T$, wobei das Erzeugen der Umgebung durch
$PlaceToEnv : Seq($MPlace$) \times$ MContext $\rightarrow$ Env

abgeleitet wird. *PlaceToEnv* sorgt dafür, dass alle an Plätze gebundene Objekte als Variablen der Belegungsfunktion β in der Umgebung unter dem Namen des Platzes zur Verfügung stehen. Um nicht zwischen verschiedenen Typen konvertieren zu müssen, nehmen wir der Einfachheit halber an, das sich der Rückgabetyp aus der OCL-Abfrage ergibt und Eingaben über Pins typkonform sind (sonst ist das Ergebnis $undefined$).

Aktionssemantik: Hauptregeln

Der ASM_{OCL}-Interpreter führt parallel mehrere Threads gleichzeitig aus. Dabei werden in jedem Schritt nicht alle existierenden einbezogen, sondern es wird zufällig eine Teilmenge aus der Menge der vorhandenen Threads bestimmt (ein Element der Potenzmenge $\mathcal{P}$(MThread)). Eine Vorbedingung ist, dass die Aktionen der ausgewählten Threads nicht im Konflikt stehen dürfen:

```
Rule MainLoop
choose p in P(MThread)
    do seq
        PreExecute(p)
        forall t in p\nonExecutable do par
            ExecuteThread(t)
        enddo
        PostExecute(p)
    enddo
endchoose

Rule PreExecute(threads)
nonExecutable := ResolveConflicts(threads)

Rule PostExecute(threads)
if  microStateReached(threads) then
    Trace := Trace ∪ Changes(threads)
endif
```

Da ASM-Verhalten die Eigenschaft hat, bei Konflikten der Regeln zu terminieren, die Aktionen in diesem Fall aber einfach *nicht* ausgeführt werden sollen, muss ein Mechanismus spezifiziert werden, der eine Konflikterkennung und -vermeidung ermöglicht. Dies leisten die dynamischen Funktionen in PreExecute:

$$ResolveConflicts : \mathcal{P}(\text{MThread}) \rightarrow \mathcal{P}(\text{MThread})$$
$$nonExecutable : \quad \rightarrow \mathcal{P}(\text{MThread})$$

ResolveConflict bestimmt für den jeweils nächsten auszuführenden Satz von Aktionen einer Menge von Threads, ob diese Aktionskombination gültig ist. Die Funktion liefert die Teilmenge der Eingabe zurück, die Konflikte verursachen würde. Ein Konflikt entsteht genau dann, wenn zwei Aktionen entweder am selben Objekt dasselbe Attribut schreiben, oder ein zu schreibendes Attribut an anderer Stelle in einer OCL-Abfrage genutzt wird, oder die Kombination im Aktionsraum ungültig ist (vgl. Def. 14). Diese zweistufige Filterung wird weiterhin genutzt, um die Zustandsübergangssemantik festzulegen. Da eine tiefergehende Betrachtung der Zustände bei Ausführungsläufen in Kapitel 4 erarbeitet wird, soll hier nur im Vorgriff abstrakt der Mechanismus der Tracebildung durch Funktionen in die Kernsemantik eingebettet werden. Zu diesem Zweck dient PostExecute und die Funktionen:

$$microStateReached : \mathcal{P}(\text{MThread}) \rightarrow \text{Boolean}$$
$$Changes : \mathcal{P}(\text{MThread}) \rightarrow \text{Change}$$

Ob ein neuer Zustand erreicht wurde (insb. unter Berücksichtigung der AtomicGroup, vgl. 3.3.8), legt *microStateReached* fest. Änderungen werden abstrakt als *Changes* an die Domäne Trace angehängt. Trace entspricht der in Definition 17 festgelegten Struktur. Für weitere Details sei auf Kapitel 3.1 und 4 verwiesen.

Die Regel ExecuteThread arbeitet die einzelnen Aktionen eines Threads ab und strukturiert sich im wesentlichen durch die verschiedenen Typen von Kontrollfluss- und Aktionsknoten:

```
Rule ExecuteThread(thread)
do seq
    if thread.currentNode is MInitialNode then
        NextNode(thread.currentNode)
    else if thread.currentNode is
    MFinalNode ∧ terminateThread(thread.currentNode) then
        MThread := MThread\{thread}
        MContext := MContext\thread.stack
        thread.currentNode := undef
    else if thread.currentNode is
    MFinalNode ∧ ¬terminateThread(thread.currentNode) then
        PassResult(thread.currentNode, top(thread))
        thread.stack := pop(thread.stack)
        thread.currentNode
          := element(thread.stack, size(thread.stack) - 1).currentNode
    else if thread.currentNode is MDecisionNode then
        let env = PlaceToEnv(InputPlaces(thread.currentNode), top(thread.stack))
        in
        let result = Eval(thread.currentNode.expression, env) in
        choose edge in OutgoingEdges(thread.currentNode) satisfying
          Eval(edge.guardExpression, env) = result
            thread.currentNode := Node(edge.target)
        endchoose
    else if thread.currentNode is MAction then
        MCreateAction(thread)
        MAssignAction(thread)
        MInvocationAction(thread)
        MQueryAction(thread)
        MIterateAction(thread)
        MInputAction(thread)
        MOutputAction(thread)
        MAtomicGroup(thread)
        NextNode(thread)
    endif
enddo
```

Neben den an dieser Stelle nicht weiter ausgeführten Regelaufrufen für Aktionen ist ersichtlich, wie der Kontrollfluss durch MInitialNode, MDecisionNode und MFinalNode festgelegt wird. Bei letzterem wird ferner das Terminieren eines Threads mit behandelt (vgl. Abschnitt 3.2.8). Die Verbindung zwischen Stapel und Voranschreiten eines Threads ist über *CurrentNode* realisiert. Diese Signatur ist gedacht, um für jeden Stapelkontext den aktuell zu verarbeitenden Aktions- oder Kontrollknoten zu liefern, d.h. für jeden Operationsaufruf lässt sich somit auch die Rücksprungadresse ermitteln:

$$\begin{aligned} currentNode : \text{MThread} &\rightarrow \text{MActivityNode} \\ &=_{def} CurrentNode(top(thread.stack)) \end{aligned}$$

Die in diesem Zusammenhang verwendete Regel *NextNode* navigiert die Transitionen unter Berücksichtigung von Schleifen (vgl. Abschn. 3.3.5) und wird wie folgt beschrieben:

```
NextNode(thread) =def
let edges = OutgoingEdges(thread.currentNode) in
if thread.currentNode is MIterationNode then
    if LoopEntry(top(thread.stack)) then
        thread.currentNode := ToSibling(edges)
    else
        thread.currentNode := ToInnerNode(edges)
    endif
else if empty(edges) then
    thread.currentNode := Parent(thread.currentNode)
else
    thread.currentNode := ToSibling(edges)
endif
```

Die Details zu den einzelnen Aktionen werden an passender Stelle durch Definitionen in Kapitel 3.1 ergänzt. Für eine Diskussion siehe Kapitel 5.2.2.

2.7 Temporale Logik

Temporale Logik ist eine Erweiterung der klassischen Aussagenlogik und wurde entwickelt um Aussagen über zeitliche Abläufe semantisch formal zu erfassen [105]. Als Vertreter der Familie der Modallogiken existiert in der Literatur nicht »die eine« Temporallogik, sondern eine Reihe von Logiken mit verschiedenen Syntaxdefinitionen sowie semantischen Interpretationen. Das wichtigste Unterscheidungsmerkmal ist das Verständnis und der Zugriff auf 'Zeit' bzw. 'Zeitpunkte' (vgl. [106]). Die gängigen Logiken führen dazu *temporale Operatoren* ein, denen eine *implizite* zeitliche Semantik innewohnt. Hierbei lässt sich hauptsächlich die Linear-Time Temporal Logic (LTL), die von einer linearen Folge von Zeitpunkten ausgeht, von der Computation Tree Logic (CTL) unterscheiden, welche von einer verzweigten (Baum-)Struktur der Zeit ausgeht. Da die Aussagemächtigkeit von LTL und CTL nur teilweise überlappt, sind weitere sogenannte Pfadquantoren nötig um beide Zeitkonzepte in einer Logik zusammenzufassen: der Übermenge Computation Tree Logic* (CTL*) (s.a. [107]). Neben diesen drei „Standardlogiken" existieren eine Reihe weiterer Ansätze und Variationen, die wir in dieser Arbeit nicht betrachten wollen (z.B. [101][108]), da der Hauptzweck eine beispielhafte Definition und Anwendung von Konzepten der Temporallogik ist, um den metamodellunabhängigen Ansatz einer dynamischen Analyse zu erforschen (s. Kapitel 4.3).

Für die dynamische Programmanalyse spielt für uns die LTL eine Hauptrolle, da sich ein Programmablauf wie in Abschnitt 2.5 gezeigt als eine lineare Kette von Zuständen auffassen lässt, über den sich temporale Aussagen prüfen lassen. Um zukunftsbezogene Aussagen zu formulieren[26] werden meist die Operatoren *always* (notiert als $\Box$), *next* (notiert als $\bigcirc$), *until* (notiert als $\mathcal{U}$) und *eventually* (notiert als $\Diamond$) eingeführt. Damit lässt sich zum Beispiel eine einfache Lebendigkeitseigenschaft eines Systems wie zum Beispiel *'alle gesendeten Nachrichten werden auch (irgendwann) empfangen'* wie folgt ausdrücken:

$$\Box(\forall msg.(send(msg) \rightarrow \Diamond(receive(msg)))) \tag{2.26}$$

und an einem Programmablauf überprüfen. Syntaktisch sind in diesem Ausdruck bereits prädikatenlogische Formeln mit $\forall$ Quantor sowie den Prädikaten *send* und *receive* enthalten, wobei letztere über einer Sorte definiert sind, die mit der Variable *msg* kompatibel ist. Ohne an dieser Stelle genauer auf die algebraischen Definitionen einzugehen sollte klar werden, dass die Variable *msg* durch den $\forall$ Quantor gebunden wird und eine Menge von Elementen identifiziert, die – wann immer das Prädikat *send* war wird – auch das Prädikat *receive* irgendwann *später* erfüllen muss. Eine striktere Eigenschaft, nämlich das jede gesendete Nachricht im unmittelbar nächsten Zeitpunkt empfangen werden muss, lässt sich durch den *next* Operator darstellen:

$$\Box(\forall msg.(send(msg) \rightarrow \bigcirc(receive(msg)))) \tag{2.27}$$

Um die Semantik von LTL-Formeln über einfache Propositionen zu fassen, werden im Allgemeinen Kripke-Strukturen verwendet (vgl. [110]). Eine Kripke-Struktur besteht dabei aus einer Menge von Zuständen, Transitionen und einer Zuordnungsfunktion, die angibt, in welchen Zuständen eine Proposition wahr ist:

Definition 23 (Kripke-Struktur) *Eine Krikpe-Struktur ist ein Quadruple* $K = (S, T, s_0, L)$ *mit*

[26] im Gegensatz zu vergangenheitsbezogene Aussagen, vgl. auch [109]

- *S einer nichtleeren Menge von Zuständen ("*states*")*
- *$T \subseteq S \times S$ der Transitionsrelation*
- *$s_0 \in S$ bezeichnet den initialen Zustand*
- *$L : S \to \Gamma$ ist die Beschriftung der Zustände mit einer Menge von Variablen $\Gamma = \{\beta_s\}_{s \in S}$.*

Eine Formel ist genau dann wahr, wenn sie bei ausgezeichneten Anfangszuständen auf allen Pfaden durch die Struktur wahr ist. Für unsere Zwecke werden wir im Folgenden den Pfad als Basis für die Semantik setzen und die Semantik von LTL-Formeln direkt an diesen koppeln. Dabei werden alle Funktionen und Prädikate über eine Zustandssequenz $\pi = s_0, s_1, s_2, \ldots$ evaluiert. Für das obige Beispiel wäre der Pfad:

$$s_0 : send(m_1) \longrightarrow s_1 : send(m_2) \longrightarrow s_2 : receive(m_2) \longrightarrow s_3 : receive(m_1)$$

ein gültiges Modell für die Formel[27] 2.26, allerdings nicht für die Formel 2.27.

Da in Kapitel 4 eine Erweiterung der OCL um temporale Konzepte erfolgt, wird in Abschnitt 2.7.1 mehrsortige Prädikatenlogik erster Stufe mit temporalen Operatoren als Referenz und zur weiteren Diskussion eingeführt.

2.7.1 Lineare Temporallogik mit Sorten

Im Folgenden soll LTL als besondere Form der prädikatenlogischen Temporallogik definiert werden, die auf den algebraischen Strukturen aus Abschnitt 2.4.2 aufsetzt. Die Syntax ergibt sich damit als Erweiterung prädikatenlogischer Formeln um temporale Operatoren (s. Abschnitt 2.4.1):

Definition 24 (LTL-Syntax) *Sei $\Sigma = (\mathcal{S}, \Omega, P)$ eine Signatur über die nichtleere Menge $\mathcal{S}$ von Sorten, Ω eine Menge von n-ären Operationssignaturen der Form $op : s_1 \ldots s_n \to s \in \Omega$ mit $s, s_1, \ldots s_n \in \mathcal{S}$ und P eine Menge von Prädikaten der Form $p : \langle t_1 \ldots t_m \rangle \in P$ mit $t, t_1, \ldots, t_m \in S$.*

Sei weiterhin $T_{\Sigma,s}(Vars)$ die Menge aller Terme *über die nichtleere Menge von Variablen $Vars$ zur Sorte $s \in S$, inklusive Terme über Funktionen. Dann stellt $LTL_\Sigma(Vars)$ die Menge der temporalen Formeln dar, die sich rekursiv definiert:*

i. Verum und Falsum sind Formeln: $\top, \bot \in LTL_\Sigma(Vars)$
ii. Die Propositionen $\neg\varphi, (\varphi \vee \psi), (\varphi \wedge \psi), (\varphi \to \psi), (\varphi \leftrightarrow \psi) \in LTL_\Sigma(Vars)$ für $\varphi, \psi \in LTL_\Sigma(Vars)$ bilden Formeln
iii. Gleichungen von Termen der selben Sorte bilden Formeln: $(t_1 = t_2) \in LTL_\Sigma(Vars)$ gdw. $t_1, t_2 \in T_{\Sigma,s}(Vars)$
iv. Prädikate über Terme bilden Formeln: $p(t_1, ..., t_n) \in LTL_\Sigma(Vars)$ mit $p : \langle s_1 ... s_n \rangle \in P$ und $t_i \in T_{\Sigma,s_i}$ für $i = 1, \ldots, n$
v. Allquantor und Existenzquantor bilden Formeln: wenn $x \in Vars$, dann $\forall x(\varphi) \in LTL_\Sigma(Vars)$ und $\exists x(\varphi) \in LTL_\Sigma(Vars)$ für $\varphi, \psi \in LTL_\Sigma(Vars)$
vi. Die Temporalquantoren 'next', 'until', 'always' und 'eventually' bilden Formeln: $\bigcirc(\varphi), \varphi \mathcal{U} \psi, \Box(\varphi), \Diamond(\varphi) \in LTL_\Sigma(Vars)$ for $\varphi, \psi \in LTL_\Sigma(Vars)$

Es sei angemerkt, dass Funktionsausdrücke in der Menge der Terme enthalten sind, temporale Operatoren aber nur Formeln erweitern und somit zu Wahrheitswerten ausgewertet werden.

[27] Man spricht auch i.Allg. von einem *Modell*, das eine temporale Formel erfüllt. Da das Wort 'Modell' in unserem Kontext schon anders gebraucht wird, wollen wir es nicht erneut überladen und vermeiden diese Bezeichnung im Folgenden wann immer möglich.

Für die Semantik nehmen wir ohne Beschränkung der Allgemeinheit an, dass die algebraischen Strukturen mit der Definition von ASM kompatibel sind und Zustandssequenzen eines ASMs über $\pi = \mathbb{S}_0 \to \mathbb{S}_1 \to \mathbb{S}_2 \to \ldots$ beschrieben sind (vgl. Abschnitt 2.5.1). Jeder durchlaufene Zustand $\mathbb{S}_t$ bildet somit einen Zeitpunkt, in dem die nicht temporalen Teilformeln ausgewertet werden können, während temporale Operatoren Aussagen über die gesamte Sequenz ausdrücken:

Definition 25 (Semantik von LTL-Formeln) *Sei $LTL_\Sigma(Vars)$ eine Menge von Temporalformeln und $\mathcal{M} = \langle \pi, S \rangle$ ein Modell. Die Gültigkeit einer Formel $\varphi \in LTL_\Sigma(Vars)$ über eine Zustandssequenz $\pi = \mathbb{S}_0 \to \mathbb{S}_1 \to \mathbb{S}_2 \to \ldots$ (abgekürzt $\pi \models \varphi$, wenn $\mathcal{S}$ sich über den Kontext ergibt) ist induktiv über die Wahrheitsfunktion der einzelnen Zustände (geschrieben als $\pi_i \models \varphi$, für den i-ten Zustand) wie folgt definiert:*

i. Verum und Falsum: $\pi_i \models \top$ *und* $\pi_i \not\models \bot$
ii. Propositionen für $\varphi, \psi \in LTL_\Sigma(Vars)$*:*
$\pi_i \models \neg\varphi$ *gdw.* $\pi_i \not\models \varphi$,
$\pi_i \models \varphi \vee \psi$ *gdw.* $\pi_i \models \varphi$ *oder* $\pi_i \models \psi$,
$\pi_i \models \varphi \wedge \psi$ *gdw.* $\pi_i \models \varphi$ *und* $\pi_i \models \psi$,
$\pi_i \models \varphi \to \psi$ *gdw.* $\pi_i \not\models \varphi$ *oder* $\pi_i \models \psi$,
$\pi_i \models \varphi \leftrightarrow \psi$ *gdw.* $\pi_i \models \varphi$ *und* $\pi_i \models \psi$ *oder* $\pi_i \not\models \varphi$ *und* $\pi_i \not\models \psi$
iii. Gleichungen: $\pi_i \models t_1 = t_2$ *gdw.* $[\![t_1]\!]^{\mathbb{S}_i}_\beta = [\![t_2]\!]^{\mathbb{S}_i}_\beta$
iv. Prädikate: $\pi_i \models p(t_1, \ldots, t_n)$ *gdw.* $[\![p(t_1, \ldots, t_n)]\!]^{\mathbb{S}_i}_\beta$ *wahr ist*
v. Allquantor: $\pi_i \models (\forall x.\varphi)$ *gdw. für alle* $a \in Vars_s$ $[\![\varphi]\!]^{\mathbb{S}_i}_{\beta[x/a]}$ *wahr ist*
vi. Existenzquantor: $\pi_i \models (\exists x.\varphi)$ *gdw. mindestens für ein* $e \in Vars_s$ *gilt* $[\![\varphi]\!]^{\mathbb{S}_i}_{\beta[x/e]}$
vii. Next-Operator: $\pi_i \models \bigcirc(\varphi)$ *gdw.* $\pi_{i+1} \models \varphi$
viii. Until-Operator: $\pi_i \models \varphi \,\mathcal{U}\, \psi$ *gdw. für mind. ein* k *mit* $i < k$ *für alle* $i \leq u < k$ *gilt* $\pi_u \models \varphi$ *und für alle* $j \geq k$ *gilt* $\pi_j \models \psi$
ix. Always-Operator: $\pi_i \models \varphi$ *gdw. für alle* j *mit* $j \geq i$ *gilt* $\pi_j \models \varphi$
x. Eventually-Operator: $\pi_i \models \varphi$ *gdw. ein* k *existiert mit* $k \geq i$ *und* $\pi_k \models \varphi$

Eine Formel φ erfüllt π, wenn sie ab dem ersten Zustand gilt: $\pi_0 \models \varphi$.

Im Vergleich zu klassischer LTL als Erweiterung der Aussagenlogik bindet diese Definition die Auswertung von Termen an verschiedene Zeitpunkte. Dadurch lassen sich Variablen mittels Quantoren und temporalen Operatoren verschieden binden, wie z.B. in den einführenden Beispielen 2.26 und 2.27 gezeigt. Diese Grundidee übertragen wir in Kapitel 4 auf Ausführungsläufe mit MActions, wobei die dortigen Zustandsübergänge einer weitergehenden Betrachtung bedürfen. Eine Diskussion zu verwandten Arbeiten erfolgt in Kapitel 5.4.

KAPITEL 3

MActions Framework

In diesem Kapitel wird die MActions-Sprache und das darauf basierende Framework zur Modellsimulation im Detail beschrieben. Mit dessen Hilfe wird die Basis geschaffen, die bisher rein strukturellen Metamodelle um eine operationale Ausführungssemantik zu erweitern und Modellsimulationen durchzuführen.

Anmerkung zur Terminologie: MActions bezeichnet die Sprache, in der die operationale Semantik für εMOF-Modelle spezifiziert wird. Da die Ausführungssemantik faktisch parametrisiert werden kann — z.B. in puncto Parallelität und durch neue Aktionen —, sprechen wir von einem *Framework*, besonders im Hinblick auf eine Implementierung.

3.1 MActions: Operationale Semantik für MOF

Die Sprache MActions (Kurzform für *MOF Actions*) stellt eine Aktionssemantik als Zusammenschluss der drei Modellierungstechnologien MOF, OCL und UML-Aktionen/Aktivitäten dar. Im Kern steht die Verbindung von MOF-Strukturelementen mit aktionsbasierten Verhaltensbeschreibungen durch Aktionen, welche als Syntax UML verwenden. Das Grundgerüst aus Aktionen, Kontroll- und Datenfluss wird hierbei aus UML übernommen, wobei jedoch alle Aktionen eine spezielle Update-Semantik für die Verarbeitung von MOF-Modellen erhalten, die auf den Konzepten zur universellen Aktionssemantik aus Kapitel 2.5.4 basiert. Anders formuliert verhalten sich MActions zu UML-Aktionen wie UML zu MOF: als Syntax zur sonst entkoppelten Semantik. OCL dient in diesem Zusammenschluss der Navigation mittels Abfragen (*engl. Queries*) über Eigenschaften der Modelle sowie zum Ausdruck von Bedingungen im Kontrollfluss.

Zur Definition der MActions verfolgen wir eine doppelte Strategie. Zum einen soll der Ansatz metazirkulär über sich selbst definiert werden, also mit MActions. Dies ist innerhalb der Metamodellierung eine verbreitete Technik und bildet das Pendant zur Strukturdefinition von MOF. Zusätzlich werden unter Rückgriff auf die algebraischen Konzepte und des ASM_{OCL}-Interpreters aus Kapitel 2 korrespondierende Definitionen angegeben. Dadurch soll nicht nur eine mathematisch formale Semantik angegeben werden, sondern eine Brücke zur dynamischen Analyse geschaffen werden.

Nach einem einführenden Beispiel wird in den folgenden Abschnitten jeweils

die abstrakte Syntax, graphische Notation, Bedingungen/Einschränkungen sowie die Semantik beschrieben.

3.1.1 Voraussetzungen zur Modellausführung

Die generelle Ausführungssemantik der MActions ist gekoppelt an zwei (Zustands)-Räume, die jedweder Modellausführung zu Grunde liegen. Der erste wird als *Abstrakter Syntaxraum* (ASR) bezeichnet und enthält ein oder mehrere Modelle desjenigen Teils eines Metamodells, der die abstrakte Syntax beschreibt. Der Abstrakte Syntaxraum ist endlich und unveränderbar. Dahingegen enthält der *Laufzeitraum* (LZR) alle Objekte und Werte die während einer Modellausführung erzeugt und verändert werden. Die Endlichkeit des Laufzeitraums hängt von der Sprachsemantik und den Modellen des Syntaxraums ab und ist in den meisten Fällen unendlich. Diese Trennung erweist sich als hilfreich für eine Aufzeichnung von Änderungen und Auswertung.

Beide Räume[1] werden durch das Konzept der *Extents* in MOF abgebildet (vgl. 2.3). Ein *Extent* bezeichnet einen adressierbaren Raum, der Modellelemente enthält. Insbesondere beschreiben beide Räume die natürliche Grenze für alle OCL-Abfragen der Form `allInstances`. Alle Modelle die nicht in einem der beiden Räume enthalten sind, werden der *Umgebung* zugeschrieben. Zum Austausch mit der Umgebung und somit dem Transfer in und aus dem LZR dienen spezielle Ein-/Ausgabe-Aktionen (vgl. Abs. 3.3.7 und 3.3.6). Zusammen bilden ASR und LZR eine Spezialisierung des Aktionsraums **AR** (s. Def. 14), bei dem die Menge von Aktionen MAct bezeichnet und deren mögliches gleichzeitiges Auftreten $\mathcal{A}$ parametrisiert werden kann:

1. Standardmäßig ist $\mathcal{A} = \mathcal{P}(\text{MAct})$, d.h. alle Aktionen sind gleichzeitig erlaubt.
2. Eine Einschränkung auf n-gleichzeitige Aktionen wird durch

$$\mathcal{A}_n = \bigcup_{i=1..n} \{\{a_1, ..., a_i\} | a_1, ..., a_i \in \mathcal{P}erm(\text{MAct})\}$$

   beschrieben, wobei $\mathcal{P}erm(\text{MAct})$ die Permutationen über MAct bezeichnet, d.h. $\mathcal{A}_n$ enthält alle Teilmengen mit bis zu n Elementen und deren Permutationen.

Die Menge $\mathcal{A}$ kann um bestimmte Aktionen und Aktionsmengen reduziert werden, um diese exklusiv nacheinander auszuführen. So würde z.B. $\mathcal{A}_n \setminus \{\{a, b\}\}$ mit $a, b \in$ MAct sicherstellen, dass a und b niemals gleichzeitig auftreten. Initial zur Modellausführung sind beide Räume mit Modellen gefüllt, die im Folgenden als *initiales Laufzeitmodell* und *abstraktes Syntaxmodell* bezeichnet werden.

Prozesse sind in MActions eingeschränkt auf Threads (s. Def. 19). Initial wird ein einzelner Thread, der *Hauptthread*, gestartet (s. 3.2.7). Dieser beginnt seine Abarbeitung und sollten keine weiteren Threads gestartet werden, so wird von der möglichen Parallelität der Aktionen faktisch kein Gebrauch gemacht.

3.1.2 Theorem der Universellen Transformationsmaschine

Unabhängig von der im Weiteren beschriebenen MActions Sprache formulieren wir zunächst einige Grundannahmen abstrakt über die Maschine, die unsere Aktionen ausführt. Die Eigenschaften und Gesetzmäßigkeiten dieses „Universums“ stellen

[1] In Bezug zu den Metaebenen könnte man synonym auch von M1- und M0-Modellräumen sprechen. Aus der Diskussion zur Relativität der Ebenen in Abschnitt 2.2 werden wir sie stattdessen den Eigenschaften der enthaltenden Modelle nach als ASR und LZR bezeichnen

den äußersten, nicht beeinflussbaren Rahmen einer operationalen Semantik dar und beeinflussen a priori die Ausführung in Bezug auf Zustände, Gleichzeitigkeit von Ereignissen und der Zeit. Die universelle Aktionssemantik aus Kapitel 2.5.4 fasst diese Axiome formal.

Als Grundannahme über die Modellausführung sehen wir neben den ASR und LZR als Grundkonzept jeder operationalen Semantik die *Transformation des Laufzeitmodells*, welche gezielt durch Aktionen gesteuert wird. Ähnlich der von A.Turing definierten Turing Maschine setzen wir im Folgenden als Basis eine abstrakte *Universelle Transformationsmaschine* (UTM), die eine operationale Semantik unmittelbar interpretieren kann und die die folgenden Axiome erfüllt:

Zustand Die UTM hat keinen eigenen Zustand, sondern verwaltet Laufzeitmodelle als einzige Zustandsgrößen. Man kann sich die UTM als Universelle Turing Maschine vorstellen, bei der das Eingabeband durch objektorientierte Modelle und die Überführungsfunktion/Zustände durch (eine endliche Anzahl von) Aktionen ersetzt wurde. Insbesondere können endlich viele Änderungen am Modell gleichzeitig durchgeführt werden[2].

Diskreditätskriterium Änderungen am Modell erfolgen diskret. Das heißt, es existieren keine „Zwischenzustände“ in denen z.B. Objekte zwar erzeugt, seine Slots aber noch nicht angelegt wurden. Jede modellverändernde Aktion führt zu einem eingeschränkt gültigen Modell. Dabei ist die Konformität zum Metamodell ein hinreichendes, aber kein notwendiges Kriterium. Zum Beispiel sind Referenzen mit einer Multiplizität von `[1..1]` nach Objekterzeugung immer undefiniert und eine Abfrage resultiert in dem Wert *Undefined*. [3]

Konsistenzkriterium Alle parallel ausgeführten Änderungen führen zu einem konsistenten, *beobachtbaren* Zustand des Aktionsraums. Dieser Zustand kann eingefroren betrachtet und aufgezeichnet werden. Es wird keine Aussage darüber getroffen, ob und wie verschiedene Aktionen gleichzeitig ausführt werden, solange gilt, dass *falls* eine Aktion ausgeführt wurde, der Folgezustand beobachtbar ist. Es finden also auch keine „versteckten“ Aktionen statt.

Kohärenzkriterium Parallele Änderungen am Modell interferieren nicht. Jede ausgeführte Aktion verhält sich exakt so, als ob keine andere Aktion parallel über dem Laufzeitmodell operieren würde. Ansonsten werden hinsichtlich parallel existierender Prozesse keine weiteren Einschränkungen gemacht. Das Kohärenzkriterium dient dem Schutz der Semantik einer jeden Aktion und schließt widersprüchliche Änderungen aus.

Zeit Zeit existiert lediglich a posteriori als Folge der Zustandsänderungen am Laufzeitmodell. Während einer Aktion „vergeht“ im eigentlichen Sinne also keine Zeit, noch existieren Zeitgeber oder Uhren per se. Zeitbegriffe wie 'vorher', 'nachher' und 'gleichzeitig' beschreiben nur kausal die Ereignisse der Zustandsänderungen als logische Relationen von beobachtbaren Zuständen (quasi 'Zeitpunkten'). Hiervon ungeachtet kann ein Metamodell Zeit als Zustandsgröße in Form einer oder mehrerer Attribute definieren, die nach und nach z.B. durch stete Inkrementierung Modellzeit ausdrücken.

Dieser Satz von Axiomen gibt den allgemeinen Rahmen vor, der noch nichts über eine konkrete operationale Semantik sagt, der jedoch unveränderliche Voraussetzungen schafft. So lassen sie operationale Semantiken mit sowohl deterministischem als

[2] s.a. Semantik in Abschnitt 3.3.8

[3] Allerdings ist das Setzen von Referenzen explizit auf `null` nicht möglich.

auch nichtdeterministischem Verhalten zu, da das Kohärenzkrit. nur „Überschneidungen“ von Aktionen ausschließt, jedoch nicht etwa Determinismus von Aktionen fordert oder das Fortschreiten von Folgeaktionen einschränkt. Dennoch adressieren die Axiome das von Börger et al. für asynchrone ASMs aufgezeigte *State Stability Problem* (vgl. [111]).

Aus dem Konsistenzkrit. und dem Diskreditätskrit. folgt unmittelbar eine lineare Sequenz von in sich konsistenten Modellzuständen, die für eine Zustandsgröße nur genau einen Wert zu einem Zeitpunkt zulassen. Dadurch wird das Aufzeichnen von linearen Traces möglich, welche die einzelnen durchlaufenen Zustände beschreiben. Da Parallelität in den Aktionen nicht ausgeschlossen ist, unterliegen aufgezeichnete Zustandsfolgen i.Allg. der partiellen (aber totalen) Ordnung aus Kapitel 2.5.4. Unter einem Trace verstehen wir vorläufig gemäß Kapitel 2.5 eine Form von globale aufgezeichneten Laufzeitmodellzuständen. Eine genauere Strukturdefinition für MOF-Modelle erfolgt in Abschnitt 4.1.3.

Vergleicht man das Kohärenzkrit. mit der Semantik der Aktionen aus Def. 15, schränken wir an dieser Stelle das allgemeine Aktionsmodell bewusst ein. Während in der universellen Aktionssemantik z.B. eine Aktion **neutralize** definiert sein könnte, die immer im Zusammenhang mit anderen Aktionen deren Effekt im Hinblick auf Modelländerungen verhindert („neutralisiert“), ist dies hier nicht mehr möglich. Dadurch erhalten MActions-Definitionen die Eigenschaft der Kompositionalität (*compositional semantics*, s.a. [37]).

Diese Grundanforderungen an eine abstrakte Ausführungsmaschine geben einen Spielraum für verschiedene operationale Semantiken und spiegeln die impliziten Annahmen wider, die den folgenden Definitionen zu Grunde liegen. Die vorgestellten MActions sind also nur *eine* mögliche Sprache zur Definition der Transformationen des Laufzeitmodells, nämlich die, die die *elementaren* Aktionen umfasst. Elementare Aktionen sind genau solche, die nur eine einzige Änderung am Modell durchführen, das heißt entweder Erzeugen von Objekten oder (Er-)Setzen von Werten/Referenzen. In Abschnitt 3.3.9 werden Erweiterungsaktionen vorgestellt, um komplexere Änderungen auszuführen. Die Relation zu anderen Sprachen ist ausführlich in Kapitel 5 diskutiert.

3.1.3 Beispiel: Hello Meta-World!

Um ein grundsätzliches Verständnis herzustellen, soll zunächst das bekannte ’Hello-World’ Programm durch MActions darstellt und erläutert werden. Abbildung 3.1 zeigt die aus zwei Aktionen bestehende Verhaltensbeschreibung. Intuitiv beginnt

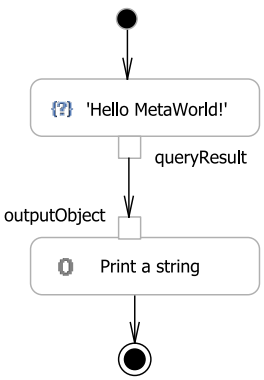


Abbildung 3.1: Das Hello World Programm als MActions

die Ausführung beim *initial node*, dem eine sog. OCL-QueryAction folgt (erkenntlich am {?} Symbol). Der Query ist hier das konstante OCL-Literal `'Hello MetaWorld!'`, welches am Ausgangspin `queryResult` zur Verfügung gestellt wird und über den Objektfluss zur nächsten Aktion weitergereicht wird. Allgemein kann ein komplexer OCL-Query an dieser Stelle stehen, dessen Resultat über den Pin bereitgestellt wird. In der folgenden OutputAction (Namens `Print a string`) steht das Literal am Eingangspin `outputObject` bereit und kann weiter verarbeitet werden. Im Beispiel erfolgt lediglich eine Ausgabe an die Umgebung[4], bevor das Verhalten mit einem *final node* endet.

In diesem einführenden Beispiel wurde auf eine Einbettung in ein Metamodell verzichtet, da es sich um eine einfache Ausgabe handelt und keine Modellstrukturen einbezogen wurden. In den folgenden Absätzen wird nun eine informelle Beschreibung des Kontroll- und Datenflusses von Aktivitäten beschrieben, inkl. statischer Semantik. Anschließend wird in Abschnitt 3.2.7 eine präzise Definition der dynamischen Semantik mittels Aktionen nachgereicht.

3.2 Aktivitäten

Eine Aktivität ist der Container für Elemente der Verhaltensbeschreibung. Sie enthält Knoten und Kanten, die je in einer Vielzahl von Spezialformen existieren und durch ihre Zugehörigkeit zur Aktivität gruppiert werden. Aktivitäten können parametrisiert sein (s. 3.2.3). Man unterscheidet zwei Formen von Aktivitäten: das *kontextlose* und das *kontextbezogene* Verhalten (*engl. context-aware behaviour*). Letzteres bezeichnen wir im Folgenden als *Operation*, da alle enthaltenen Aktionen im Kontext eines Objekts ausgeführt werden.

Abstrakte Syntax

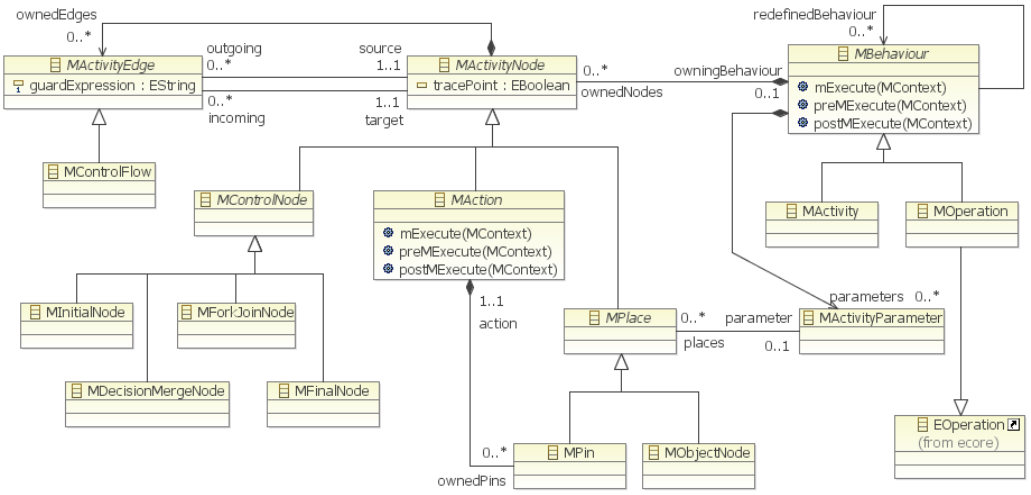


Abbildung 3.2: Übersicht: Knoten, Aktionen, Verhalten und Kontrollfluss der MActions

Das erweiterte MOF-Meta-Metamodell ist in Abbildung 3.2 dargestellt. In der abstrakten Klasse `MBehaviour` wird die Grundstruktur einer Aktivität über enthal-

[4] vgl. Abschnitt 3.3.6

tene Knoten (`ownedNodes`) und Kanten (`ownedEdges`) festgelegt. Dabei wird Verhalten durch die konkreten Klassen `MActivity` und `MOperation` definiert. Instanzen von `MActivity` repräsentieren kontextloses Verhalten und werden, wie Klassen, in Packages platziert. Im Gegensatz dazu steht das kontextbezogene Verhalten der `MOperation`, bei dem Aktionen im Kontext eines Objekts ausgeführt werden, weshalb diese Klasse die in MOF als Platzhalter vorgesehene Klasse `Operation` spezialisiert. Konsequenterweise ist innerhalb eines Operationsaufrufs das Objekt mit dem OCL-Ausdruck `self` in allen enthaltenen Aktionen adressierbar.

Graphische Notation

Zur graphischen Notation von `MOperations` wird die in MOF für Operationen vorgesehene textuelle Darstellung der Signatur innerhalb einer Klasse verwendet (vgl. 3.2), für `MActivities` die in der UML-Spezifikation beschriebene Notation für Aktivitäten (s. Abb. 3.3).

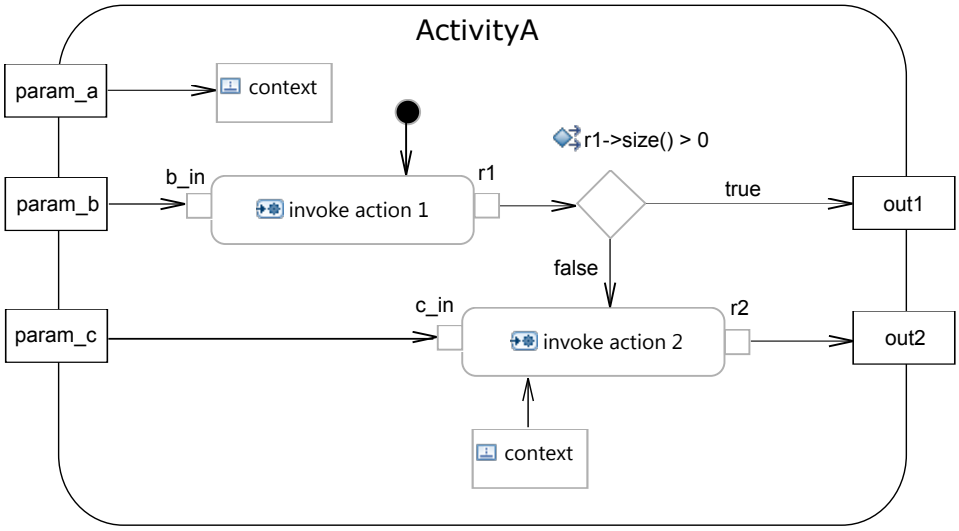


Abbildung 3.3: Notation für `MActivity` mit Parametern

Die beispielhafte Abbildung zeigt neben den Parametern auch den Datenfluss zu Objektplätzen. Dabei wird neben den Eingangsparametern `param_a`, `param_b` und `param_c` der Ausgabeplatz `out2` nur belegt, wenn der Bedingungsknoten mit *true* ausgewertet wird. (Im anderen Fall ist der Wert des Rückgabeparameters `out2` *oclUndefined*.)

In den meisten Abbildungen der folgenden Abschnitte wird dieser Aktivitäts-„Rahmen“ um Aktionen weggelassen und lediglich der eigentliche Aktionsfluss dargestellt, da der Kontext durch die Bildunterschriften angegeben wird.

3.2.1 Kontrollfluss

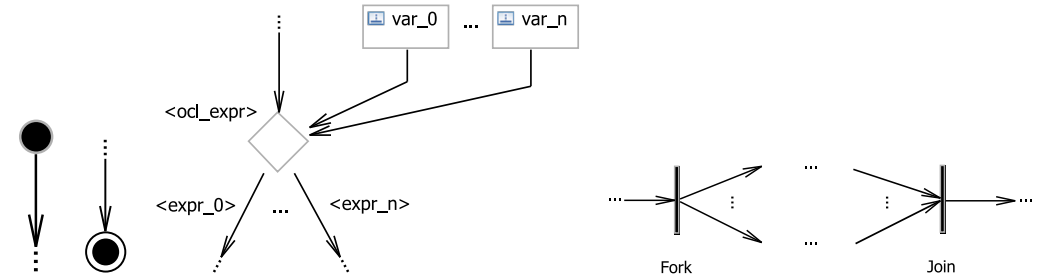
Der Kontrollfluss ist strukturell über die Verkettung von Knoten mittels Kanten ausgedrückt. Kanten sind gerichtet, d.h. eine Kante besitzt genau einen Vorgängerknoten (`source`-Referenz) und genau einen Nachfolgerknoten (`target`-Referenz). Der Kontrollfluss schreitet immer von Vorgänger zum Nachfolgerknoten voran.

Falls ein Knoten mehr als eine ausgehende Kante hat, um Datenfluss an verschiedene Nachfolgerknoten auszudrücken, muss definiert werden, welche der Kanten im nächsten Schritt verarbeitet wird, ansonsten ist das Verhalten *undefiniert*. Diese

Kennzeichnung erfolgt durch das Schlüsselwort `'main'` an der Kante, die den Kontrollfluss festlegt.

3.2.2 Kontrollknoten

Neben Aktionen als Spezialisierung der allgemeinen Klasse `Node` existieren eine Reihe von Kontrollknoten zur Steuerung des Kontrollflusses (s. Abb. 3.2.2. Diese werden im Folgenden kurz mit entsprechenden Bedingungen dargestellt.



(a) Initial Node (b) Final Node (c) Decision Node (d) Fork/Join Kontrollfluss

Abbildung 3.4: Sprachmittel zum Kontrollfluss (Notation)

Initial Node, Final Node

Zwei spezielle Knoten sind der *Initial Node* und der *Final Node* (Klassen `MInitialNode` und `MFinalNode`). Diese markieren den Anfang bzw. das Ende eines Ausführungsblocks. Deshalb besitzt der Initialknoten keine eingehenden Kanten und der Finalknoten keine ausgehenden Kanten. Während es in einer Aktivitätsdefinition nur einen `MInitialNode` geben darf, können mehrere `MFinalNode` existieren:

```
MInitialNode.allInstances()->forAll(incoming->isEmpty()) and
MFinalNode.allInstances()->forAll(outgoing->isEmpty())

MBehaviour.allInstances()->forAll(ownedNodes->one(n :
MActivityNode | n.oclKindOf(MInitialNode)))
```

Zusätzlich gibt es für Endknoten zwei spezielle Semantiken: Für einen Endkonten mit dem Flag `terminateThread = true` wird bestimmt, dass nicht nur der aktuelle Fluss, sondern auch der aktuelle *Thread* bei Erreichen beendet wird (s.a. 3.2.8, 3.3.3). Für einen Endknoten innerhalb einer Iteration trägt er die bekannte `break`-Semantik von Schleifen: der Kontrollfluss setzt sich beim Nachfolgerknoten der Iteration fort.

Fork und Join Knoten

Eine Menge von Aktionen, bei denen die Ausführungsreihenfolge keine Rolle spielt, kann lokal zu einer Aktivität parallel ausgeführt werden. Zu diesem Zweck dient ein `Fork`-Knoten der Aufsplittung in mehrere, parallele Flüsse. Diese enden entweder mit einem `MFinalNode` oder werden über einen korrespondierenden `Join` synchronisiert zusammengeführt.

Zusätzlich stellt eine `Fork-Join`-Verzweigung eine Möglichkeit dar, n-aus-m-Kontrollflüssen nichtdeterministisch auszuwählen. Wird also z.B. bei einer 3-pfadigen

Verzweigung $n = 2$ gewählt, werden 2-aus-3 Pfaden nichtdeterministisch ausgewählt und parallel ausgeführt. Somit ist der Fall, dass alle Pfade gleichzeitig ausgeführt werden, der Sonderfall bei dem $n = m$.

DecisionNode

Bedingter Kontrollfluss wird durch Entscheidungsknoten (Klasse `MDecisionMergeNode`) ausgedrückt. Dazu muss eine Bedingung als OCL-Ausdruck am Knoten angegeben werden (Attribut `expression`), welche gegen Zielwerte an den ausgehenden Kanten verglichen wird. Diese Zielwerte der ausgehenden Kanten sind wiederum durch OCL als `guardExpressions` angegeben. Sollten die Kantenbedingungen nicht disjunkt sein, so ist das Verhalten nichtdeterministisch. Im Fall das keine Kante über einen Wert verfügt, der der ausgewerteten Bedingung entspricht, ist das Verhalten `Undefined`, wenn der Entscheidungsknoten in einer Aktivität/Operation enthalten ist. Als Spezialfall für Entscheidungsknoten innerhalb von Iterationen setzt sich jedoch der Kontrollfluss am Anfang der Iteration mit dem nächsten Wert fort. Letzteres ist also quasi ein `continue` für Iterationsschleifen. Zudem kann ein Decision/Merge-Knoten auch zur Zusammenführung (*engl. Merge*) verschiedener Kontrollflüsse dienen.

3.2.3 Datenfluss, Aktivitätsparameter

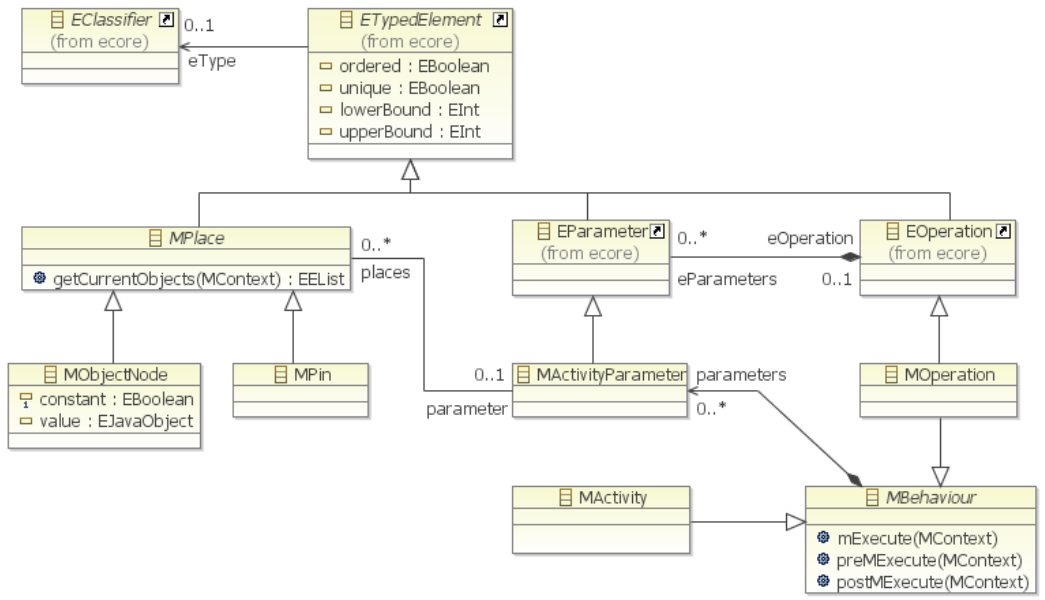


Abbildung 3.5: Parameter, Datenfluss und Typen

Datenfluss wird explizit durch Verbindung von Pins und/oder Objektknoten ausgedrückt. Daten sind immer Objekte und werden ausschließlich per Referenz weitergereicht (vgl. Instanzmodell in Abschnitt 3.2.5). Beim Datenfluss wird syntaktisch zwischen Pins und Objektknoten unterschieden, wobei letztere hauptsächlich zur Darstellung von Aktivitätsparametern dienen. Semantisch sind beide äquivalent. Dennoch können Objektknoten auch zur Übersichtlichkeit eingesetzt werden, in dem *derselbe* Objektknoten des abstrakten Syntaxmodells mehrfach in der graphischen Notation vorkommt.

Durch die Bindung von Objektknoten an Parameter kann jede Aktivität parametrisiert werden. Diese Bindung erfolgt durch Setzen der `places/parameter`-Referenz (vgl. Abb. 3.2). Es können beliebig (endlich) viele Eingabe- und Ausgabeparameter spezifiziert werden. Falls ein Aufruf eines Verhaltens nicht alle Parameter belegt, erhalten diese als Defaultwert, sofern nicht anders angegeben, *OclUndefined*. Das gleiche gilt für Pins von Aktionen: sollte ein Pin nicht durch vorher ausgeführte Aktionen belegt werden, z.B. weil eine Vorgängeraktion durch verzweigten Kontrollfluss nicht ausgeführt wurde, so wird der Pin mit *OclUndefined* belegt. Details zum Kontrollfluss sind in Abschnitt 3.2.7 beschrieben.

Bedingungen zur statischen Semantik

Zusätzlich zu den im Metamodell definierten Attributen definieren wir die folgenden abgeleiteten Attribute (derived properties), die den Zugriff und die Navigation durch MAction-Graphen erleichtern:

1. Eingangspins lassen sich durch folgende Attribute abfragen:

   derive MActivityNode::ownedInputPins : **Set**(MPin) =
   self−>collect(ownedPins.outgoing−>isEmpty())
   MActivityNode::ownedInputPlaces : **Set**(MPin) = self.ownedInputPins

2. Ausgangspins sind durch folgendes Attribut beschrieben:

   derive MActivityNode::ownedOutputPins : **Set**(MPin) =
   self−>collect(ownedPins.incoming−>isEmpty())

3. Iterationsaktionen enthalten einen Pin für die Variable des OCL-Iterationsausdrucks. Das heißt, wenn `var` die Variable des Ausdrucks beschreibt, dann gilt die folgende Bedingung:

   derive self−>ownedPins−>includes(varPin : MPin | varPin.name = 'var')

   Da der Name dieses Pins variabel ist, fügen wir zusätzlich ein abgeleitetes Attribut für den Zugriff ein:

   derive MIterateAction::iterateVar : MPin =
   self.ownedPins−>select(pin : MPin | pin.name = 'var')

4. Der erste Knoten der Iterationsaktion kann durch folgendes Attribut abgefragt werden:

   derive MIterateAction::firstNode : MActivityNode =
   self.ownedNodes−>select(self.outgoing.name = 'main').target

5. Alle eingehenden und ausgehenden Kanten eines Knoten, d.h. eigene und die der enthaltenen Pins erhält man durch die Attribute:

   derive MDecisionNode::outgoingEdges : **Set**(MActivityEdge) =
   self.outgoing−>union(self.ownedOutputPins−>collect(outgoing))
   derive MDecisionNode::incomingEdges : **Set**(MActivityEdge) =
   self.incoming−>union(self.ownedInputPins−>collect(incoming))

6. Alle mit einem Entscheidungsknoten verbundene Eingangsplätze beschreibt das Attribut:

   derive MDecisionNode::inputPlaces : **Set**(MPlace) =
   self.incoming−>collect(edge : MActivityEdge | edge.source.oclIsKindOf(MPlace))

7. Der initiale Knoten eines Verhaltensgraphen kann direkt mit dem Attribut:

   derive MBehaviour::initialNode : MInitialNode =
   self.ownedNodes−>select(node : MActivityNode | node.oclIsTypeOf(MInitialNode))

abgefragt werden.

8. Der Nachfolgerknoten einer Aktion kann wie folgt abgefragt werden:

```
def MAction::getNextNode() : MActivityNode =
  if (not self.outgoingEdges.target.oclIsUndefined()) then
    self.outgoingEdges.target
  else if (self.owningBehvaiour.eContainer().oclIsKindOf(MActionGroup)) then
    self.owningBehvaiour.eContainer().oclAsType(MActionGroup)
  else
    null
  endif
```

Damit lassen sich eine Reihe von statischen Bedingungen an ein MACTIONS-Modell stellen, die die gültigen Instanzen des Metamodells weiter einschränken. Jede Modellausführung beginnt mit einer kontextfreien `MActivity`.[5] Meist wird darin entweder ein existierendes Objekt gesucht oder ein neues Objekt angelegt, dessen Verhalten wiederum gerufen wird.

```
MActivity.allInstances()->size() > 0
```

Somit ist der globale Einstiegspunkt in eine MACTIONS-Simulation frei wählbar und eine Sache der Implementierung, wie die Kennzeichnung/Übergabe erfolgt.

Die schaltende Kante wird mit dem Schlüsselwort `main` gekennzeichnet, um den Kontrollflusses von mehreren möglichen Datenflüssen zu unterscheiden.

```
if (self.outgoingEdges->size() == 1) then
  self.outgoingEdges->any()
else
  self.outgoingEdges->one(edge : MActivityEdge | edge.name = 'main')
endif
```

3.2.4 Redefinition

Operationen können in abgeleiteten Klassen redefiniert werden. Die Redefinition ermöglicht die Beschreibung von *Interpreter Entwurfsmustern* (vgl. [112]) bei der Definition einer Semantik und wurde bei den MACTIONS selbst angewandt. Entgegen objektorientierten Programmiersprachen, bei denen die Überschreibung in der Regel per Konvention durch denselben Namen ausgedrückt wird, referenziert eine überschreibende MACTIONS-Operation immer die redefinierte Vorlage (im Metamodell durch `redefinedBehaviour`). Für Ein- und Ausgabeparameter verlangen wir lediglich Typkonformität, d.h. Ausgabeparameter können kovariant und Eingabeparameter kontravariant verändert werden. Dies sichert folgende statische Bedingung zu:

```
context MOperation inv:
if (not self.redefinedBehaviour.oclIsUndefined()) then
  self.parameter->forAll(p : EParameter |
    if (p.direction = #INOUT) then
      p.type = baseParam(p).type
    else if (p.direction = #IN) then
      p.type = baseParam(p).type or baseParam(p).type.eAllSupertypes()->includes(p.type)
    else if (p.direction = #OUT) then
      p.type = baseParam(p).type or p.type.eAllSupertypes()->includes(baseParam(p).type)
    endif)
endif
```

[5] vgl. `public static void main()` Methode von Java oder C++

```
def: baseParam(op : MOperation, p : EParameter) : EParameter
  = op.redefinedBehaviour.parameter->select(name = p.name)
```

Im Prinzip entspricht das Auflösen und dynamische Binden zur Laufzeit anhand der Operationssignatur der klassischen Inklusionspolymorphie unter Berücksichtigung von kontravarianten Eingabeparametern und kovarianten Rückgabeparametern. Jedoch kann durch Redefinition das von normaler Mehrfachvererbung bekannte *Diamant-Problem* auftreten, da bei jeder Invokationsaktion die Operationsredefinition in der Klasse des Laufzeitobjekts gerufen wird und dort u.U. mehrere Redefinitionen vererbt wurden. Diese Art von Konflikten räumen wir durch eine zusätzliche Eindeutigkeitsbedingung aus:

```
context EClassifier inv:
self.eOperations->forAll(op | not self.eOperations->exists(other |
    redefined(op).intersection(redefined(other))))

def: redefined(op : MOperation) : Set(MOperation)
  = if (op.redefinedBehaviour.oclIsUndefined())
      Set{}
    else
      redefined.union(op.redefinedBehaviour)
    endif
```

Mit anderen Worten wird gefordert, dass in einer Klasse bei keinen zwei redefinierten Operationen eine gemeinsame Basis überschrieben wird (unabhängig vom Vererbungspfad).

3.2.5 Laufzeitmodell

Das Laufzeitmodell der MActions basiert auf dem orthogonalen Instanzmodell aus Kapitel 2, Abschnitt 2.3.2. Ergänzt zu dieser generischen Objektstruktur der Modelldaten stellt das Laufzeitmodell alle Konzepte bereit, die die Aufrufe von Aktivitäten ausmachen: Threads, Stapel und Stapelkontexte, sowie Variablen- bzw. Platzbelegungen (s. Abb. 3.6). Die hier dargestellten Klassen sind direkt der auf

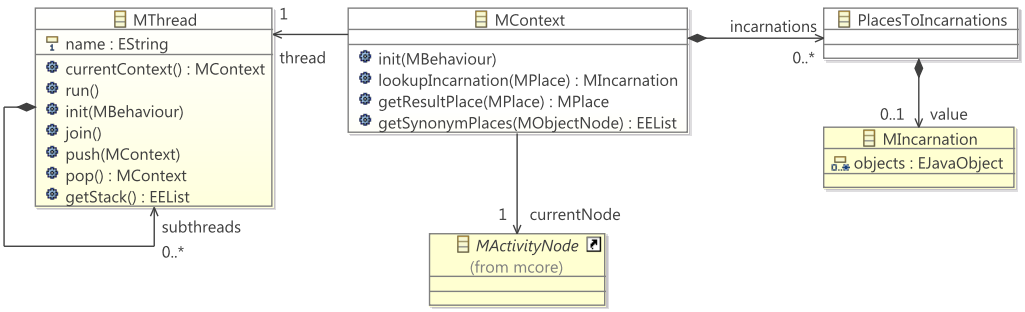


Abbildung 3.6: Das Laufzeitmodell des MAction Interpreter

EMF basierten Implementierung entnommen, wie an den Primitiven- und Listentypen erkennbar ist. Der aktuell ausgeführte Aktivitätsknoten wird mittels `currentNode` referenziert. Der Stapel eines Threads ist über die Operation `#getStack` realisiert, der Zugriff auf Objekte eines Stapelkontextes durch `MIncarnation`. Für die Spezifikation verwenden wir im Folgenden eine modifizierte Variante des direkten Zugriffs mittels der Referenzen `stack` und `contexts`, die ebendiese Beziehungen ersetzen.

Die Metaklasse *PlacesToIncarnations* und die Typisierung der Objekte `objects` als *EJavaObjects* sind ebenfalls Implementierungsdetails, die sich aufgrund der Objektverwaltung ergeben haben und nur der Vollständigkeit halber angeführt sind.

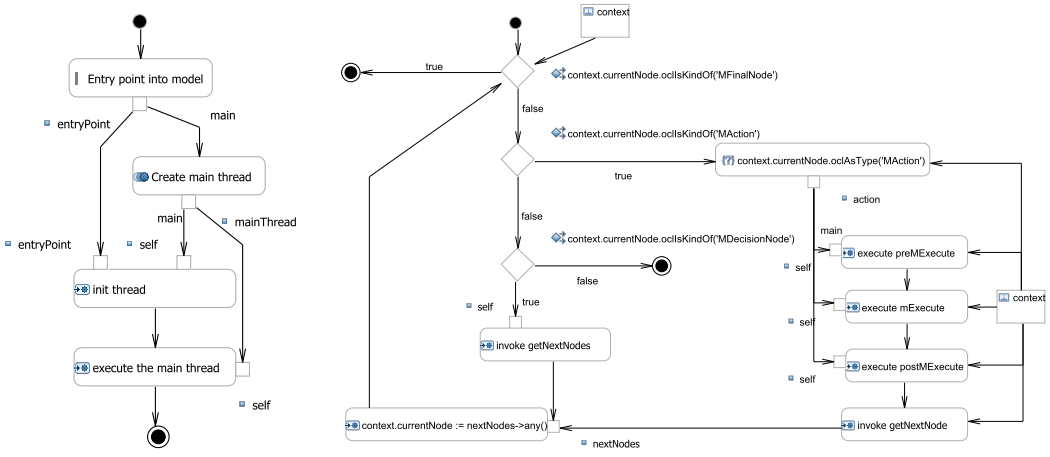
3.2.6 Horizont des Laufzeitraums

Der Laufzeitraum stellt den beobachtbaren Rahmen (Horizont) für die Ausführung dar. Mit dem initialen Laufzeitmodell und der im Weiteren definierten CreateAction (s. 3.3.1) und InputAction (s. 3.3.7) werden Objekte in diesem Raum erzeugt oder hinzugefügt, die standardmäßig für den Rest der Ausführung darin verbleiben – d.h. es verschwinden keine Objekte, die nicht mehr über Plätze/Referenzen im Laufzeitmodell referenziert werden (etwa ähnlich eines *garbage collections*). Diese Eigenschaft nennen wir *Objektpersistenz*. Objektpersistenz wird immer dann merklich, wenn mittels der Operation `allInstances()` nach Instanzen einer Metaklasse gefragt wird.

In den während dieser Arbeit erstellten Metamodellen mit MActions ergaben sich keinerlei Probleme bezogen auf die Objektpersistenz, da die meiste Zeit über explizite Referenzen auf Laufzeitobjekte verwiesen und navigiert wurde. Dennoch soll an dieser Stelle zumindest vorgeschlagen werden, dass die Objektpersistenz aufgegeben werden könnte, in dem man z.B. an der Menge `allInstances()` mit `remove` ein Objekt „löschen“ könnte.[6]

3.2.7 Ablaufsemantik

Der Kontrollfluss schreitet generell von einem Knoten zu dessen Nachfolger voran oder wird durch die vorgenannten Kontrollflussknoten beeinflusst. Initiiert wird ein Ablauf durch die Aktivität `Startup`, die in Abb. 3.7(a) dargestellt ist.



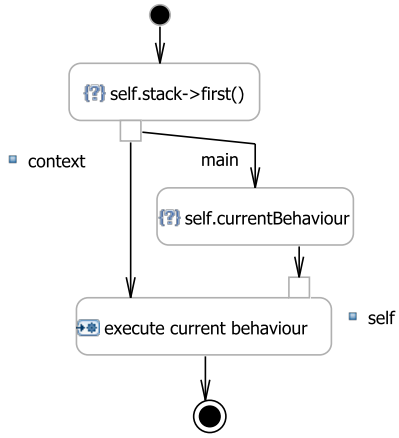
(a) Startup MActivity (b) MBehaviour::mExecute

Als Eingabe wird zunächst der Einstiegspunkt als `entryPoint` erfragt, bevor der Hauptthread gestartet wird. Jede Ausführung läuft somit innerhalb eines `MContext`-Objekts eines Threads ab, der mittels `init` initialisiert und mittels `run` gestartet wurde (s.Abb. 3.7(d) und 3.7(c)).

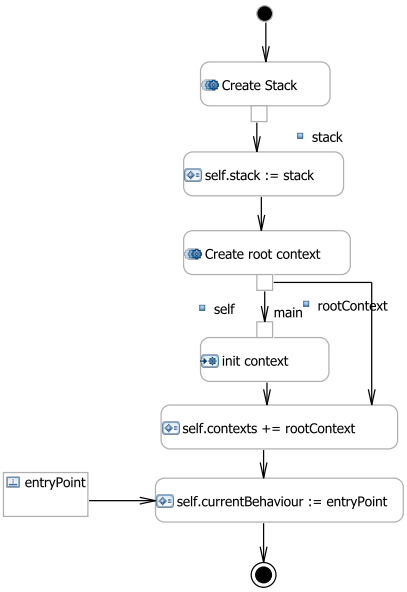
Die Hauptablaufsemantik einer Aktivität besteht in `mExecute` (Klasse `MBehaviour`) aus einer Schleife, die iterativ jeden Knoten abarbeitet und bei Aktionen jeweils de-

[6] Etwaige Auswirkungen für die dynamische Analyse gemäß Kapitel 4 wären zu untersuchen

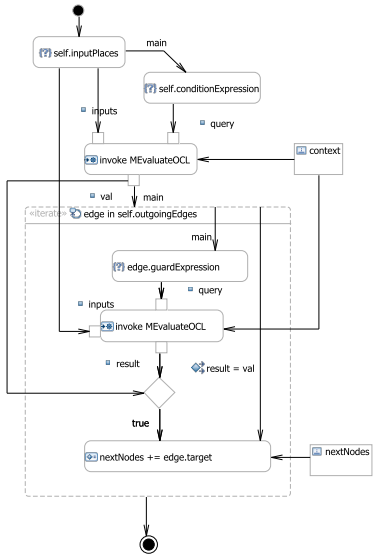
ren Aktivität aufruft (s. Abb. 3.7(b)). Entscheidungen die den Kontrollfluss steuern werden in der Hilfsaktivität `getNextNodes` ausgewertet (s. Abb. 3.7(e)). Dort steckt die eigentliche Evaluierung des Bedingungsausdrucks und die Verzweigung zur jeweiligen ausgehenden Kante.



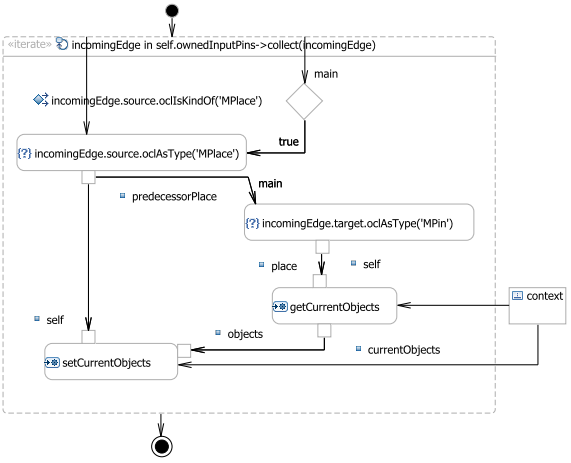
(c) MThread::run



(d) MThread::init



(e) MBehaviour::getNextNodes



(f) MBehaviour::mPreExecute

Der Datenfluss findet parallel zum Kontrollfluss statt und bedarf keiner separaten Modellierung im Aktionsfluss. Im Falle von mehreren ausgehenden Datenflusskanten ist der Kontrollfluss mit `main` gekennzeichnet. Bevor eine Aktion ausgeführt wird, werden Objektreferenzen von allen Plätzen vorwärts zu Nachfolgeplätzen propagiert. Es existiert nur die Übergabe per Referenz bei allen Datentypen: wertebasierte Weitergabe meß explizit z.B. durch OCL beschrieben werden. Spezifiziert ist der Datenfluss in `mPreExecute`, siehe Abb. 3.7(f).

Initialisierung der ASM-Semantik

Die Kernsemantik der MActions wurde bereits in Kapitel 2.6.2 erläutert. Für die äquivalent ASM-Initialisierung der Maschine mit einem Hauptthread wird die folgende Ausgangssituation angenommen:

```
main: → MThread
root: → MContext
domain MThread =def {main}
domain MContext =def {root}
domain Trace =def {}

Rule Init
main.stack := push(root, Stack(MContext))
currentNode(main) := InitialNode(entryPoint)
```

Analog zur Abb. 3.7(d) definieren wir die Thread-Initialisierung wie folgt (für PassParameters s. Abschnitt 3.3.3):

```
InitThread(thread) ≡
  let root = GenOid(MContext),
  newStack = Stack(MContext) in
    MContext := MContext ∪{root}
    thread.stack := newStack
    newStack := push(root, newStack)
    PassParameters(thread.currentNode, root)
```

3.2.8 Parallelität

Echte Parallelität, also wahrhaftig zeitgleich ablaufende Aktionen, werden durch *Threads* beschrieben. Wie in Kapitel 2.5.4 erläutert kann ein Thread nur eine Aktion gleichzeitig ausführen. In der Terminologie lehnen wir uns historisch nicht an 'Prozess' (oder 'Agent') an, da ein Thread wörtlich genommen nur eine Aktion gleichzeitig ausführen kann und somit der in vielen Programmiersprachen realisierten Threads eher entspricht. Im Gegensatz allerdings etwa zu Threads in Java oder dem POSIX-Standard können `MThreads` nur gestartet aber niemals vom Aufrufer gestoppt oder beendet werden. Das Thread-Ende tritt ausschließlich durch einen Finalknoten mit gesetztem Flag `terminateThread` einer Aktionsfolge ein, die der Threads selbst abarbeitet.

Zwei parallel laufende Threads sind in der Abarbeitung ihrer Aktionen lediglich durch Berücksichtigung des Konsistenz- und Kohärenzkriteriums im Aktionsraum eingeschränkt, d.h. atomare Aktionen an verschiedenen Objekten können parallel stattfinden, während Aktionen an denselben Objekten nacheinander ausgeführt werden (s.a. 2.5.3 und 2.5.4). In beiden Fällen ist insbesondere die Reihenfolge von einem globalen Standpunkt zufällig. Als Folge müssen Synchronisierungen und Determinismus der Ausführungsreihenfolge, so fern gewünscht, immer explizit modelliert werden. Das Gleiche gilt für *Fairness*. Dies kann entweder durch die Modellierung von sequentiellen Aktionsfolgen innerhalb eines Threads geschehen, oder bei mehreren Threads durch Nutzung von speziellen Threadaktionen (vgl. Abschnitt 3.3.8). Das starten eines Threads kann auf zwei Arten geschehen:

1. Aufruf einer Aktivität oder Operation mittels einer Invokationsaktion, bei der das Flag `startThread` auf *true* gesetzt ist. In diesem Falle beginnt der Aufruf

in einem neuen Thread, der sich komplett entkoppelt weiterentwickeln kann. Der aufrufende Thread setzt seine Arbeit unmittelbar fort, Rückgabewerte zwischen den Threads sind nicht möglich, lediglich der Austausch über gemeinsam genutzte Objekte.

2. Mittels einer Fork-Join-Kombination (oder alternativ mit Finalknoten) lassen sich parallele *Unterkontrollflüsse* beschreiben. Ein verzweigter Fork-Join-Fluss beschreibt dabei eine von Unterthreads abgearbeitete Folge von Aktionen, bei der der initiierende Thread mit der Ausführung bis zur fertigen Abarbeitung (d.h. dem Join-Knoten) wartet. Ein einfaches Zusammenführen der Rückgabewerte der Unterthreads wird durch gemeinsam genutzte Plätze ermöglicht.

Die zweite Form des Threadstartens mittels Fork lässt sich im Prinzip auf die erste Form durch Modelltransformation reduzieren, wobei die Kontrollflusszweige zu Aktivitäten umgeformt werden müssen, die von separaten Threads abgearbeitet werden, während der initiierende Thread mit der Ausführung wartet.[7] Dabei müssen ggf. für gemeinsam genutzte Plätze Aktivitätsparameter eingeführt werden. Da in puncto Parallelität durch Fork-Join keine echte Erweiterung beschrieben wird, sondern lediglich eine Vereinfachung in der Handhabung von Threads und deren Synchronisation, wurde bei der Semantikdefinition der MActions auf eine getrennte Spezifikation verzichtet.

[7] Die Semantik für MAtomicGroups wird hierbei allerdings schwieriger

3.3 Aktionen

Aktionen sind atomare Einheiten der Verhaltensbeschreibung zur Transformation des Laufzeitmodells. Es können keine zwei Aktionen gleichzeitig (innerhalb eines *Extents*) ausgeführt werden, die in einem Konflikt stehen, d.h. das Resultat ist bei bekannter Eingabe immer determiniert (s. Konsistenzkriterium in 3.1.2). Aktionen tauschen ihre Daten über *Pins* aus. Im Allgemeinen können Aktionen eine Vielzahl von Pins enthalten, bei denen Objekte über Eingangspins erhalten, dann transformiert oder navigiert werden und das Resultat an Output-Pins weitergereicht wird. Somit entsteht zusätzlich zum Kontrollfluss ein expliziter Datenfluss. Zur Visualisierung wird die UML-Notation für Aktionen verwendet (Abgerundetes Rechteck für Aktionen mit angrenzenden Quadraten für Pins). Komplexere Aktionen lassen

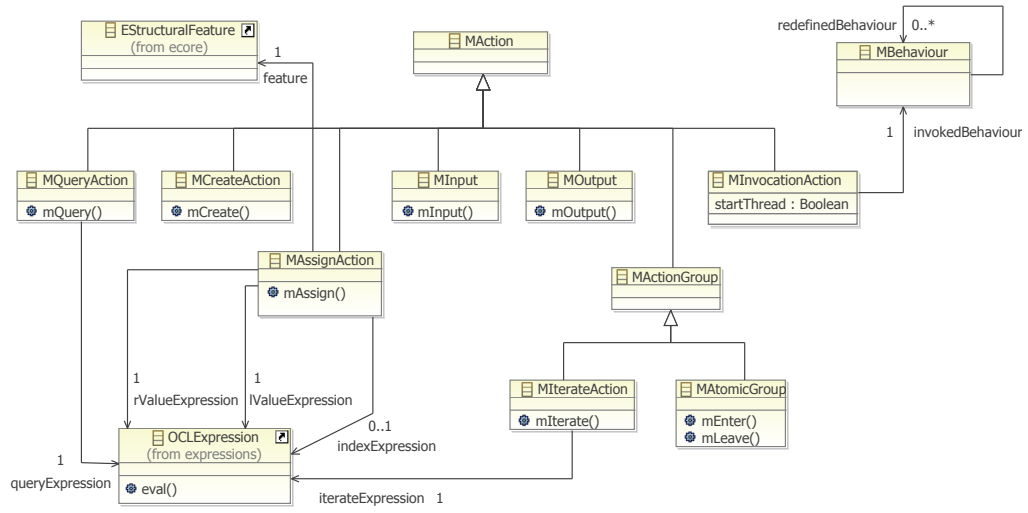


Abbildung 3.7: Übersicht: Abstrakte Syntax der Aktionen und die Relation zum OCL-Metamodell

sich entweder mittels Aktivitäten gruppieren und/oder durch MAtomicGroup zu einer (neuen) unteilbaren Einheiten verschmelzen (vgl. 3.3.8). Um die verschiedenen Aktionen voneinander zu unterscheiden, werden zusätzlich Symbole am linken Rand der Syntax der Aktion notiert. Im Folgenden wird die Syntax und Symbole zu jeder Aktion jeweils mit formaler Semantik in ASM definiert.

3.3.1 Create Action

Die CreateAction stellt den Beginn des Lebenszyklus eines jeden Objekts dar. Sie erzeugt und initialisiert Instanzen von Klassen und definiert somit die physikalische Instanziierung von MOF-Klassen.

Während der Instanziierung wird ein neues Objekt innerhalb des Laufzeitraums erzeugt und mit Slots für alle strukturellen Merkmale seiner Klasse und aller Superklassen (rekursiv) versehen. Slots mit Defaultwerten werden entsprechend vorbelegt. Optional kann über den Eingangspin `metaObject` ein Objekt für die logische Instanziierung angegeben werden, welches fortan die Rolle des Metaobjekts für die neu erzeugte Instanz spielt.

Abstrakte Syntax

Die Aktion wird durch die Klasse `MCreateAction` definiert, welche eine direkte Ableitung von `MAction` ist (s. Abb. 3.7). Die zu instanziierende Klasse wird statisch über die `type` Referenz des enthaltenen Output-Pins festgelegt. Sowohl der Bezeichner der Aktion als auch die der Pins sind frei wählbar.

Bedingungen

1. Es muss ein getypter Pin für das neu zu erzeugende Objekt existieren:

   self.pins−>one(p : MPin | **not** p.outgoing−>isEmpty() **and not** self.type.oclIsUndefined())

2. Um das neue Objekt in eine logische Instanziierungsbeziehung zu stellen, kann bei der Erzeugung ein weiterer Pin fürs Metaobjekt angegeben werden:

   self.pins−>size() = 1 **or** self.pins−>one(p : MPin | **not** p.incoming−>isEmpty())

Graphische Notation

Die graphische Notation der `MCreateAction` ist in Abb. 3.8 dargestellt.

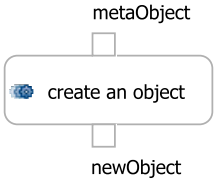


Abbildung 3.8: Create Action

Semantik

Die Semantik `MCreateAction` ist in Abb. 3.9 dargestellt.

Semantik in ASM

Rule MCreateAction(thread)
if *thread.currentNode* **is** MCreateAction **then**
  let cl = Classifier(first(InputPlaces(thread.currentNode))
    newObj = GenOid(cl) **in**
  Class(cl) := Class(cl) ∪{*newObj*}
  if *cl.metaClass* ≠ ⊥ **then**
    Meta(cl) := Meta(cl)
    ∪{(*newObj*,CurrentObjects(select(InputPlaces(thread.currentNode), 1)))}
  endif
  CurrentObjects(top(thread.stack), first(OutputPlaces(thread.currentNode))) :=
  newObj
endif

3.3.2 Assign Action

Wertzuweisungen an Attribute und Referenzen werden durch AssignActions beschrieben. Abhängig von der Multiplizität eines Features existieren verschiedene Operatoren für die Manipulation von Slots. So können einzelne Werte oder Objekt(-listen) an einem Slot gesetzt, hinzugefügt, ersetzt oder entfernt werden.

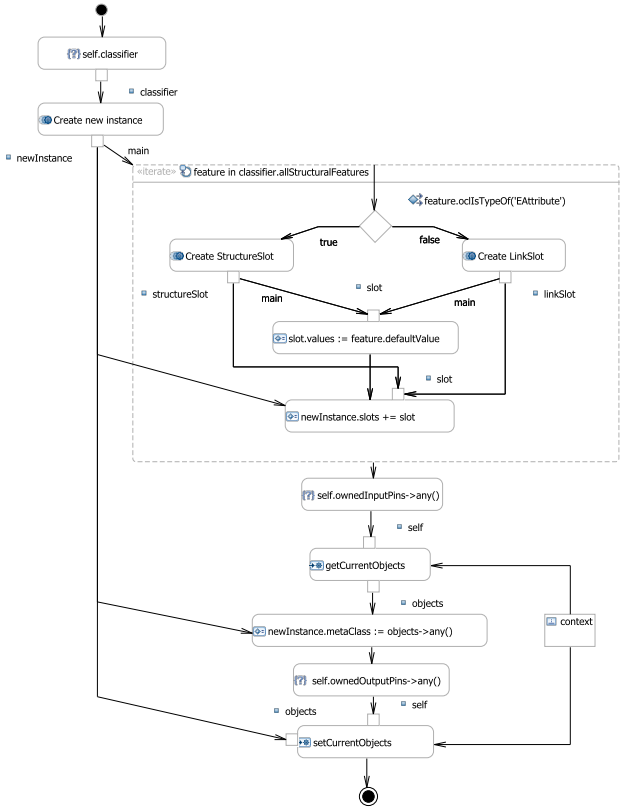


Abbildung 3.9: MCreateAction::mCreate

Abstrakte Syntax

Die Klasse `MAssignAction` als Subklasse der `MAction` modelliert die Zuweisungsaktion (s. Abb. 3.7). Um das Setzen von Werten zu ermöglichen, wird im Allgemeinen das Feature, die Art der Änderung (`kind`) und ein Ausdruck des zu setzenden Wertes benötigt (`rValueExpression`). Während der R-Wert der Zuweisung ein beliebiger OCL-Ausdruck sein kann, der einen Wert liefert, muss die `lValueExpression` das zu aktualisierende Feature beschreiben. Deshalb muss letztere in einen OCL-FeatureCallExp Ausdruck münden, dessen referenziertes Feature zum R-Wert passt (vgl. [17], Abs. 8.3.1).

Folgende Arten sind durch die Aufzählung `MAssignmentKind` unterstützt:

- `SINGLE_VALUE_ASSIGN` Zuweisen eines Einzelwerts
- `MULTI_VALUE_ADD` Hinzufügen eines Einzelwerts. Ist das zu veränderte Merkmal des Objekts mit `isUnique` gekennzeichnet, wird der Wert nur hinzugefügt, wenn er noch nicht in der Menge enthalten ist
- `MULTI_VALUE_ADD_ALL` Hinzufügen einer Menge von Werten
- `MULTI_VALUE_REMOVE` Entfernen eines Einzelwerts. Wenn der Wert nicht vorhanden ist, wird die Menge nicht verändert
- `MULTI_VALUE_REMOVE_ALL` Entfernen einer Menge von Werten.
- `MULTI_VALUE_RETAIN_ALL` Entfernen von Werten, die nicht in der gegebenen Menge enthalten sind
- `MULTI_VALUE_ADD_AT` Fügt einen Einzelwert an einem bestimmten Index in die Menge ein. Für diesen Operator muss die `indexExpression` angegeben sein und einen gültigen Index liefern (ansonsten ist das verhalten *undefiniert*)

- `MULTI_VALUE_ADD_ALL_AT` Fügt eine Menge von Werten an einem bestimmten Index ein (setzt wiederum eine gültige `indexExpression` voraus)
- `MULTI_VALUE_SET` Ersetzt die komplette Wertmenge durch den R-Wert der `rValueExpression`
- `MULTI_VALUE_REMOVE_AT` Entfernt einen Einzelwert an einem bestimmten Index

Bedingungen

1. Das angegebene Feature muss Veränderungen gestatten.

   not self.feature.oclUndefined() **and** self.feature.isChangable = true

2. Der Typ des Features muss kompatibel zum Rückgabewert des R-Werts sein unter Berücksichtigung von Multiplizitäten (Collection Types).

Graphische Notation

Die Notation für die `MAssignAction` für einwertige Features ist in Abbildung 3.10 dargestellt. Für mehrwertige Attribute und Referenzen sind folgende Notationen zu

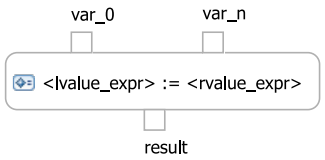


Abbildung 3.10: Assign Action

verwenden:

- `MULTI_VALUE_ADD`:
  `<feature_expr> += <r_value_expr>`
- `MULTI_VALUE_ADD_ALL`:
  `<feature_expr> += <r_value_expr>`
- `MULTI_VALUE_REMOVE`:
  `<feature_expr> ->remove( <r_value_expr> )`
- `MULTI_VALUE_REMOVE_ALL`:
  `<feature_expr> ->removeAll( <r_value_expr> )`
- `MULTI_VALUE_RETAIN_ALL`:
  `<feature_expr> ->retainAll <r_value_expr>`
- `MULTI_VALUE_ADD_AT`:
  `<feature_expr> ->add( <r_value_expr>, <index_expr>)`
- `MULTI_VALUE_ADD_ALL_AT`:
  `<feature_expr> ->addAllAt( <r_value_expr>, <index_expr>)`
- `MULTI_VALUE_SET`:
  `<feature_expr> := <r_value_expr>`
- `MULTI_VALUE_REMOVE_AT`:
  `<feature_expr> ->removeAt( <r_value_expr>, <index_expr>)`

Semantik

Die Semantik `MAssignAction` ist in Abb. 3.11 dargestellt.

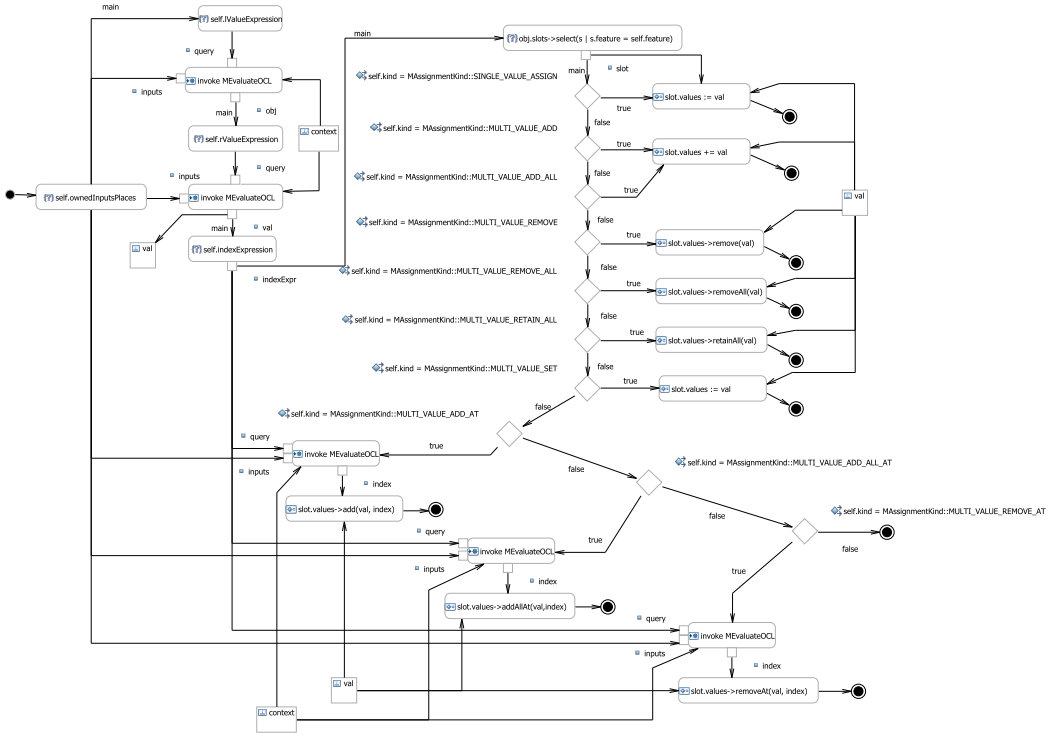


Abbildung 3.11: MAssignAction::mAssign

Semantik in ASM

Rule MAssignAction(thread)
if *thread.currentNode* **is** MAssignAction **then**
let env = PlaceToEnv(InputPlaces(thread.currentNode), top(thread.stack)) **in**
let inP = InputPlaces(thread.currentNode),
indexExpr = thread.currentNode.indexExpression,
att = AttToChange(thread.currentNode),
value = Eval(thread.currentNode.assignExpression, env),
kind = thread.currentNode.kind
in
if $isSingleValueAssign(kind) \vee isMultiValueAssign(kind)$ **then**
att := value
else if $isMultiValueAssignAll(kind)$ **then**
att := $\cup_{Seq}$(value, att)
else if $isMultiValueAddAt(kind) \vee isMultiValueAddAllAt(kind)$ **then**
att := $\cup_{Seq}$(value, att, Eval(indexExpr, env))
else if $isMultiValueRemove(kind)$ **then**
att := remove(value, att)
else if $isMultiValueRemoveAt(kind)$ **then**
att := remove(value, att, Eval(indexExpr, env))
else if $isMultiValueRemoveAll(kind)$ **then**
att := removeAll(value, att)
endif
endif

3.3.3 Invocation Action

Eine Aktivität kann weitere Aktivitäten durch InvocationActions aufrufen. Dazu wird ein neuer Ausführungskontext auf dem Stack geschaffen, der sämtliche Werte aller Pins während der Ausführung[8] verwaltet und der initial mit den Eingabewerten aller Eingangspins der Aktion selbst vorbelegt ist.

Abhängig von der Art des aufzurufenden Verhaltens, wird bei kontextbezogenen Operationen die tatsächlich auszuführende Aktivität durch Auswertung der Vererbungs- und Redefinitionsbeziehung eines `self`-Parameters (bzw. dessen Klasse) bestimmt. Eine Besonderheit der Aktivitätspolymorphie im Vergleich zu gängigen objektorientierten Programmiersprachen ist jedoch das explizite modellieren der Redefinitionsbeziehung zwischen Operationen, welches bei Mehrfachvererbung die Eindeutigkeit des auszuführenden Verhaltens sicherstellt (vgl. Abschnitt 3.2.4).

Abstrakte Syntax

Die abstrakte Syntax der `MInvocationAction` legt das aufzurufende Verhalten durch die Referenz `invokedBehaviour` fest (vgl. Abb. 3.7). Im Falle einer `MActivity` bezeichnet diese Referenz unmittelbar das auszuführende Verhalten, während bei einer `MOperation` der Typ des zur Laufzeit anliegenden Objekts am `self`-Pin darüber entscheidet, welche Operation aufgerufen wird. Dazu wird der Klassifizierer des `self`-Objekts auf vorhandene Redefinitionen der referenzierten Operation hin untersucht, das aufgelöste Verhalten dynamisch gebunden und angestoßen.

Das Attribut `startThread` bestimmt den Threadkontext des neuen Verhaltens. Ist der Wert dieses Attributs *true*, so wird ein neuer Thread inklusive Stack erzeugt und das auszuführende Verhalten parallel gestartet (vgl. Abschnitt 3.2.8). Ist der Wert `false` (Defaultbelegung), setzt der aktuelle Thread seine Ausführung im aufzurufenden Verhalten mit neuem Stackkontext fort.

Bedingungen

1. Jeder Aufruf einer `MOperation` verlangt ein Kontextobjekt `self`

   ```
   self.ownedPins->exists(contextPin : MPin | contextPin.name = 'self')
   ```

2. Für jeden Parameter einer Aktivität muß ein passender Pin vorhanden sein

   ```
   self.ownedPins->forAll(pin : MPin | self.invokedBehaviour.parameters->exists(param |
     param.name = pin.name and param.type = pin.type
       and param.multiplicity = pin.multiplicity))
   ```

Graphische Notation

Die graphische Notation der Invokationsaktion ist in Abb. 3.12 dargestellt.

Semantik

Die Semantik der `MInvokeAction` ist in Abb. 3.13 dargestellt.

Das auflösen von redefiniertem Verhalten geschieht dynamisch zur Laufzeit und ist in Abb. 3.14 dargestellt.

[8] Die Ausführung einer Aktivität bezeichnen wir auch als *Inkarnation* der Aktivität.

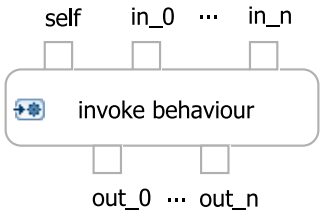


Abbildung 3.12: Invocation Action

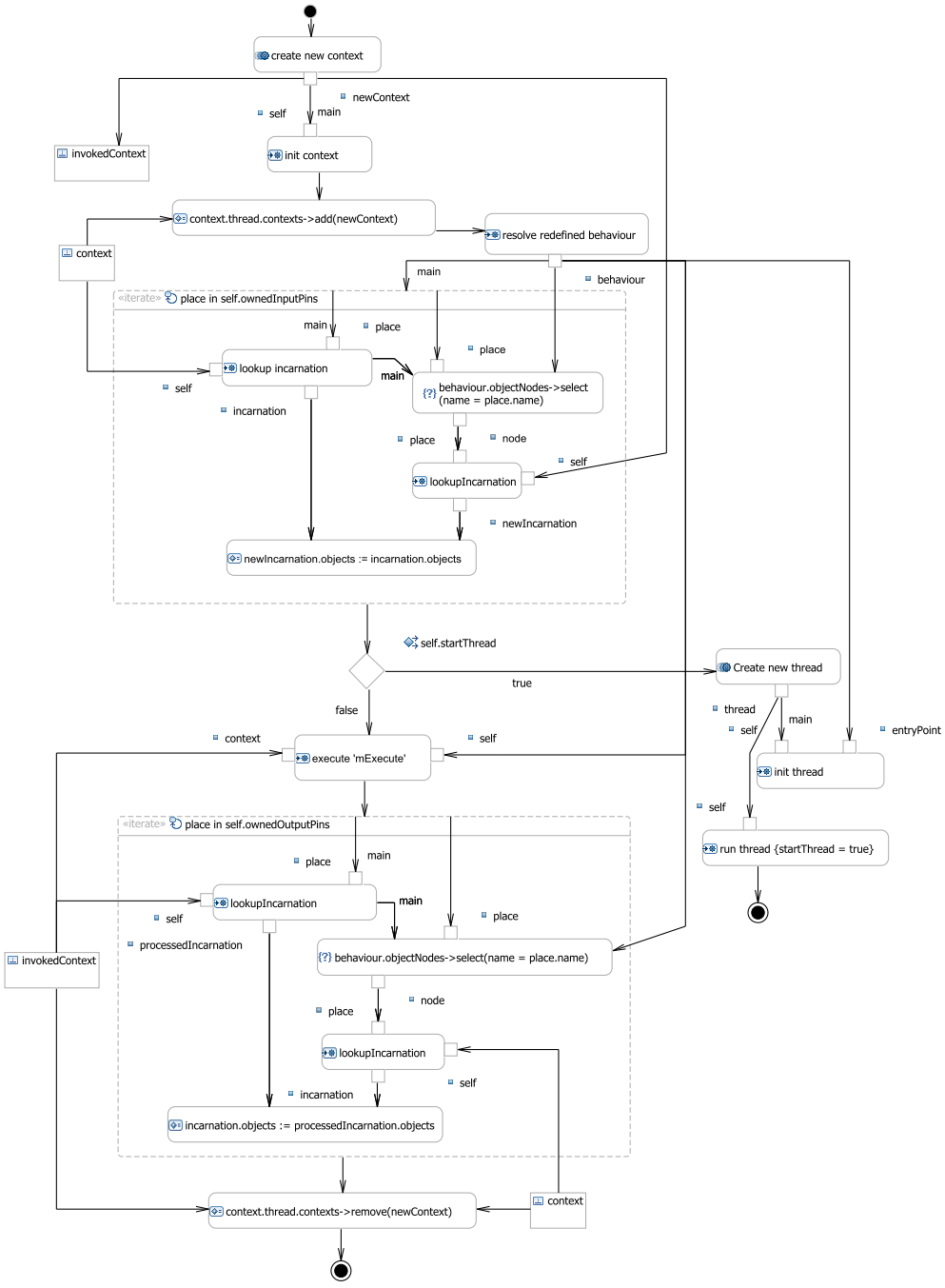


Abbildung 3.13: MInvokeAction::mInvoke

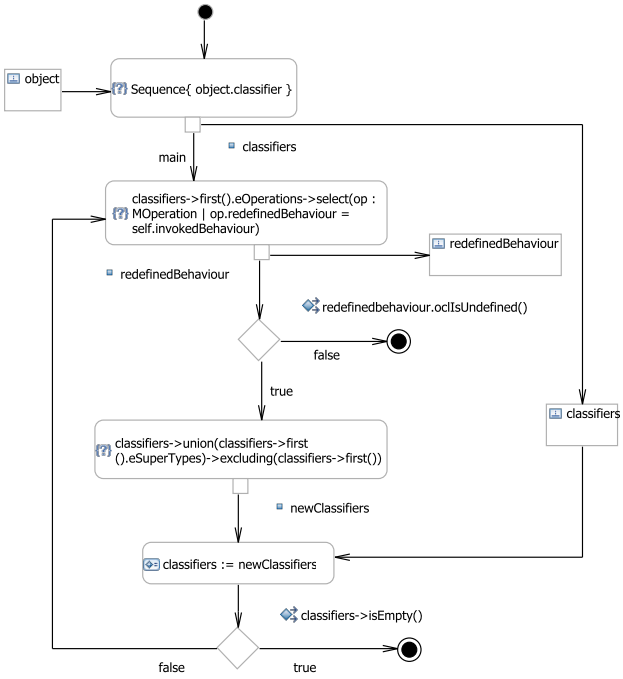


Abbildung 3.14: MInvokeAction::resolveRedefinition

Semantik in ASM

Rule MInvocationAction(thread)
if *thread.currentNode* **is** MInvocationAction **then**
  if $thread.currentNode.startThread = false$ **then**
    let newContext = GenOid(MContext) **in**
    MContext := MContext $\cup\{newContext\}$
    thread.stack := push(newContext, thread.stack)
    PassParameters(thread.currentNode, newContext)
    thread.currentNode := InitialNode(ResolveBehaviour(top(thread.stack), thread.currentNode))
  else
    let newThread = GenOid(MThread) **in**
    MThread := MThread $\cup\{newThread\}$
    InitThread(newThread)
  endif
endif

PassParameters($actNode, context$) $\equiv$
foreach $place \in InputPlaces(actNode)$ **do**
  CurrentObjects(context, ObjectPlace(place)) := CurrentObjects(context, place)
end

PassResult(actNode, thread) $\equiv$
let callContext = element(thread.stack, size(thread.stack) - 1),
context = top(thread.stack) **in**
foreach $place \in OutputPlaces(actNode)$ **do**
  CurrentObjects(callContext, ObjectPlace(place)) := CurrentObjects(context, place)
end

3.3.4 Query Action

Abfrage von Attributwerten sowie Navigation entlang Referenzen erfolgt durch QueryActions. Für die Abfragesyntax werden alle OCL-Konstrukte unterstützt, die einen Wert liefern.[9] Eine QueryAction läuft innerhalb eines Evaluierungskontextes ab, bei dem mehrere Variablen durch Eingangspins mit Werten belegt werden[10]. Durch die atomare Eigenschaft der Aktionen ist sichergestellt, dass eine Abfrage nicht mit Objekterzeugungen oder Wertzuweisungen interferiert.

Das Resultat einer Abfrage ist ein Wert, der an einem Output-Pin hinterlegt wird. Gültige Werte sind neben primitiven Werten, Objekt(-mengen), Tupeln etc. auch *OclUndefined*. Dieser repräsentiert auch den Wert `null` im Falle von nicht gesetzten Objektreferenzen.

Abstrakte Syntax

Die abstrakte Syntax der Query besteht im Wesentlichen aus einem OCL-Ausdruck, der durch die Referenz `queryExpression` bestimmt wird (Verbindung ins OCL-Metamodell, s. Abb. 3.7). Neben der Möglichkeit, Variablen über Pins in einen Ausdruck einzubeziehen, steht bei Operationen über die implizite Variable `self` immer auch das Kontextobjekt zur Verfügung.

Bedingungen

1. Es muss exakt einen Ausgangspin existieren, an dem das Resultat der Auswertung bereitgestellt wird:

   $\text{self.ownedOutputPins}{->}\text{size}() = 1$

2. Alle nicht freien Variablen des OCL-Ausdrucks müssen durch Eingangspins beschrieben werden.
3. Der Typ des OCL-Ausdrucks muss kompatibel mit dem Ausgangspin sein.

Graphische Notation

Abfragen werden in der Standardsyntax für Aktionen notiert und können mehrzeiligen Text enthalten, der den OCL-Ausdruck umfasst (s. Abb. 3.15).

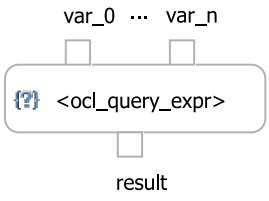


Abbildung 3.15: Query Action

Semantik

Die Semantik der `MQueryAction` verteilt sich auf die Aktion selbst (dargestellt in Abb. 3.16) sowie eine MActivity zur generischen Auswertung einer OCLExpression (dargestellt in Abb. 3.17).

[9] ausgeschlossen sind somit z.B. der `pre` Operator sowie OCL-Messages
[10] vgl. Evaluierung in der OCL-Spezifikation [17]

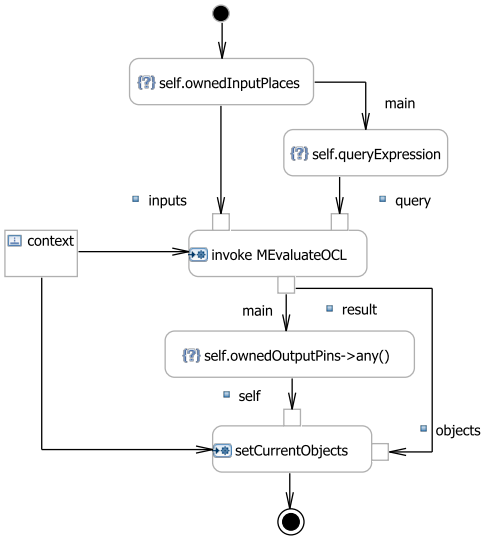


Abbildung 3.16: MQueryAction::mQuery

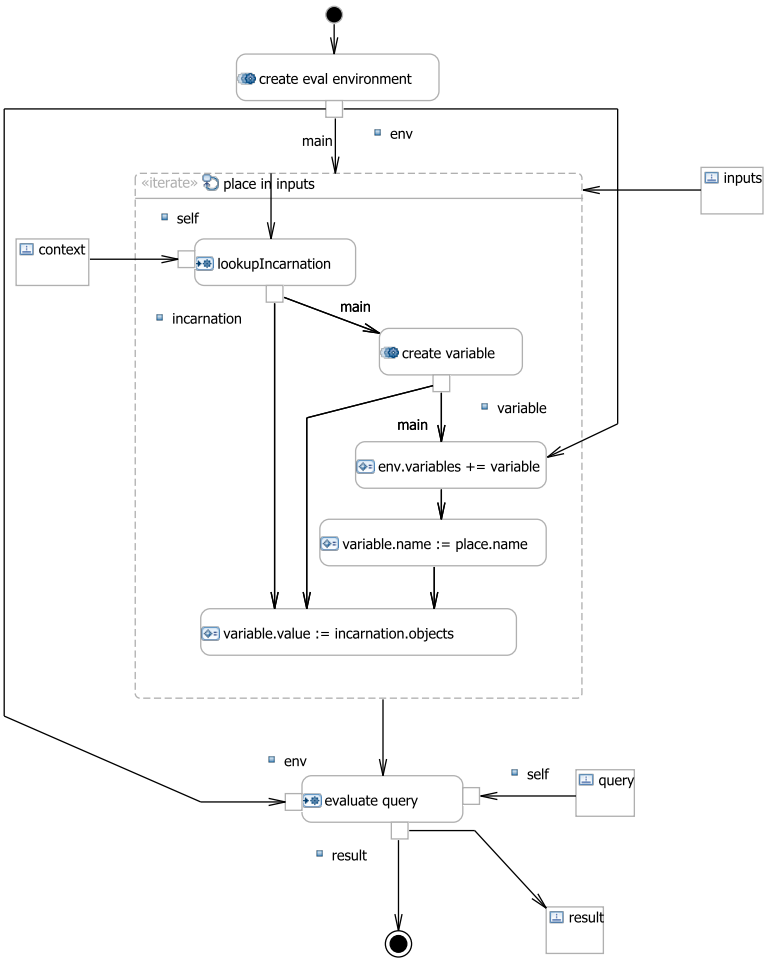


Abbildung 3.17: MEvaluateOCL

Semantik in ASM

Rule MQueryAction(thread)
let env = PlaceToEnv(InputPlaces(thread.currentNode), top(thread.stack)) **in**
if *thread.currentNode* **is** MQueryAction **then**
    CurrentObjects(top(thread.stack),
      first(OutputPlaces(thread.currentNode))) :=
    Eval(thread.currentNode.queryExpression, env)
endif

3.3.5 Iterate Action

Zur Verarbeitung von Mengen stehen Iteratoren in Form der IterateAction zur Verfügung. Die in ihr enthaltenen Aktionen werden pro Element einer gegebenen Eingangsmenge abgearbeitet. Bei jedem Durchlauf liefert die IterateAction das jeweils aktuelle Element an alle Eingangspins enthaltener Aktionen, die über eine Transition direkt verbunden sind. Die zu iterierende Menge von Elementen bestimmt sich durch eine OCL-Abfrage, die bei Aktionsbeginn einmal ausgeführt wird.

Abstrakte Syntax

Die Klasse `MIterateAction` definiert hauptsächlich die Referenz `iterateExpression` für die OCL-Abfrage zur Bestimmung der Iterationsmenge (s. Abb. 3.7). Durch Vererbung von der Klasse `MActionGroup` kann die IterateAction selbst weitere Aktionen durch `memberNodes` enthalten, welche bei jeder Iteration durchlaufen werden.

Es sei angemerkt, dass Datenfluss von Pins außerhalb der IterateAction (z.B. von vorher berechneten Werten) zu Aktionen innerhalb durchaus möglich ist, sofern die Bedingungen aus Abschnitt 3.2.3 erfüllt sind.

Bedingungen

Für diesen Ausdruck gelten ähnliche Einschränkungen wie für die QueryAction (vgl. Abs. 3.3.4):

Bedingungen

1. Alle nicht freien Variablen des OCL-Ausdrucks müssen durch Eingangspins beschrieben werden.
2. Der Typ des OCL-Ausdrucks muss kompatibel mit dem Pins und Plätzen der inneren Aktionen sein.

Graphische Notation

IterateActions werden mit gestrichelter Linie notiert, bei der die Kopfzeile die Abfrage zur Bestimmung der zu iterierenden Mengen enthält und alle enthaltenen Aktionen innerhalb des Aktionsrandes eingezeichnet werden (s. Abb. 3.18).

Semantik

Die Semantik der `MIterateAction` ist in Abb. 3.19 dargestellt.

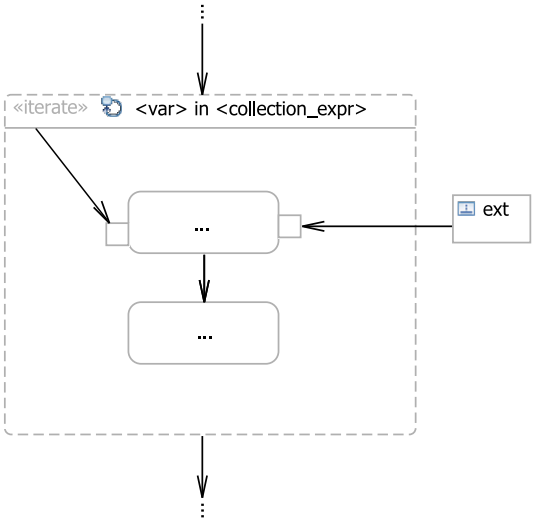


Abbildung 3.18: Iterate Action

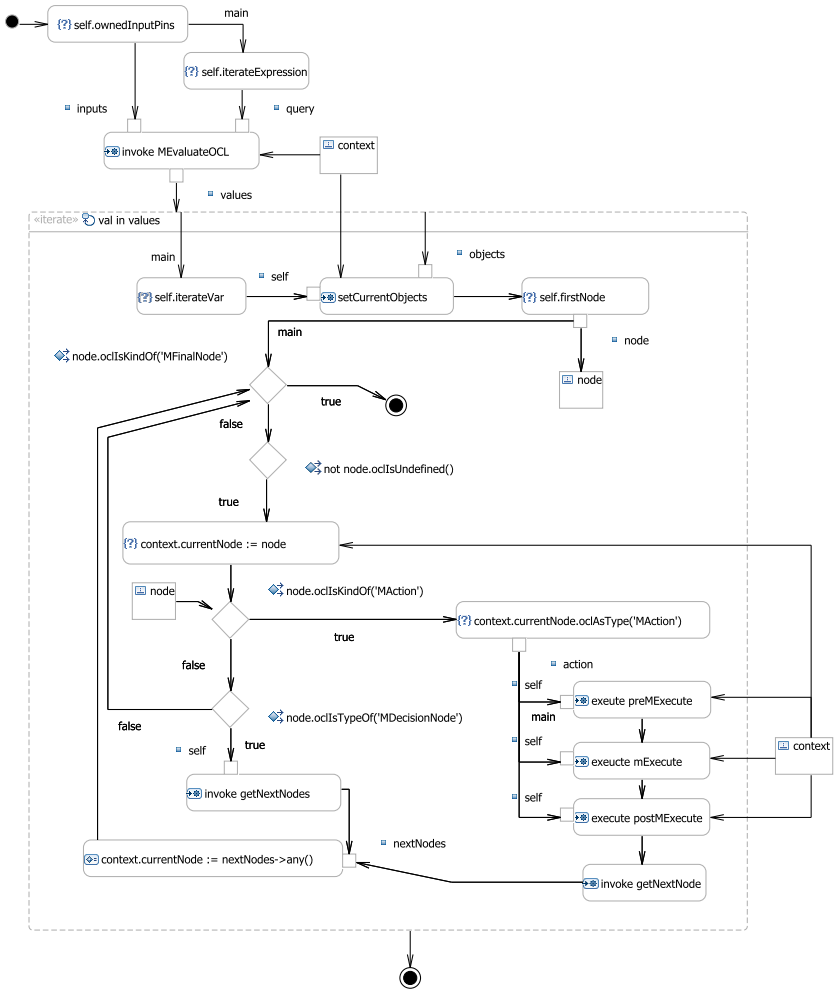


Abbildung 3.19: MIterateAction::mIterate

Semantik in ASM

```
Rule MIterateAction(thread)
if thread.currentNode is MIterateAction then
    let env = PlaceToEnv(InputPlaces(thread.currentNode), top(thread.stack)),
      elements = IterationElements(thread.currentNode, top(thread.stack)) in
    if LoopEntry(top(thread.stack)) then
        do seq
            elements := Eval(thread.currentNode.queryExpression, env)
            if size(elements) > 0 then
                forall p in ConnectedInnerPlaces(thread.currentNode) do par
                    CurrentObjects(top(thread.stack), p) := first(elements)
                enddo
                elements := remove(first, elements)
            endif
        enddo
    else
        if size(elements) > 0 then
            forall p in ConnectedInnerPlaces(thread.currentNode) do par
                CurrentObjects(top(thread.stack), p) := first(elements)
            enddo
            elements := remove(first, elements)
        endif
    endif
endif
```

Für die Iteration über Elemente in Schleifen sind neben den eigentlichen Updates zwei dynamische Funktionen definiert:

$$IterationElements : \text{MIterateAction} \times \text{MContext} \rightarrow Seq(\text{Object})$$
$$LoopEntry : \text{MContext} \rightarrow \text{Boolean}$$

IterationElements verwaltet die aktuell zu traversierenden Objekte pro Stapelkontext/Iterationsaktion, während *LoopEntry* anzeigt, ob ein Schleifeneintritt stattfand. In diesem Fall wird die Abfrage der zu traversierenden Elemente neu gesetzt.

3.3.6 Output Action

Ausgaben an die Umgebung erfolgen mit Hilfe der OutputAction. Diese Aktion stellt zusammen mit der InputAction die Schnittstelle zur Umgebung dar, indem alle an Eingabepins anliegenden Objekte an die Umgebung gereicht werden (s.a. Abs. 3.3.7). Sämtliche Ausgabeaspekte wie Formatierung und Darstellung sind nicht Teil der MActions Sprache und deshalb implementierungsspezifisch.

Abstrakte Syntax

Die Klasse `MOutputAction` definiert diese Aktion (s. Abb. 3.7).

Bedingungen

1. OutputActions können nur Eingabepins enthalten:

```
self.ownedOutputPins->isEmpty()
```

Graphische Notation

Kenntlich gemacht werden OutputActions mit einem `O` Symbol wie in Abb. 3.20 gezeigt.

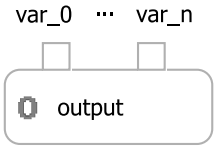


Abbildung 3.20: Output Action

Semantik

Die Semantik der OutputAction ist in Abb. 3.21 dargestellt.

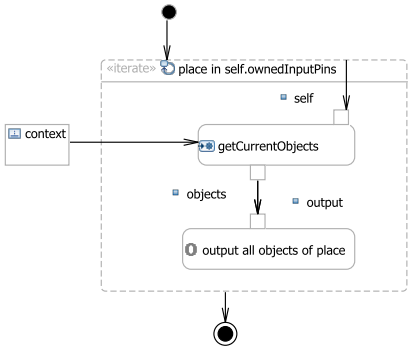


Abbildung 3.21: MOutputAction::mOutput

Semantik in ASM

Rule MOutputAction(thread)
if *thread.currentNode* **is** MOutputAction **then**
  forall *p* **in** *InputPlaces(thread.currentNode)* **do par**
    WriteOutput(CurrentObjects(top(thread.stack), p))
  enddo
endif

controlled *WriteOutput*(*place*)

3.3.7 Input Action

Eingaben erfolgen durch InputActions. Entgegengesetzt der OutputAction bildet diese Aktion die Eingabeschnittstelle zwischen Umgebung und Laufzeitraum (s.a. Abs. 3.3.6). Übertragen werden können nicht nur primitive Datentypen, sondern jedwede Objektstrukturen, d.h. (Teil-)Modelle, die konform zum jeweiligen Metamodell sind.

Abstrakte Syntax

Die abstrakte Syntax der InputAction ist definiert durch die Klasse `MInputAction` (s. Abb. 3.7).

Bedingungen

1. Eine InputAction kann nur Ausgangspins enthalten, an dem Eingabemodelle nach Aufruf der Aktion bereitstehen.

   self.ownedInputPins−>isEmpty()

Graphische Notation

Notiert wird die InputAction mit einem `I` Symbol wie in Abb. 3.22 dargestellt.

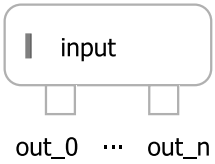


Abbildung 3.22: Input Action

Semantik

Die Semantik der `MInputAction` ist in Abb. 3.23 dargestellt.

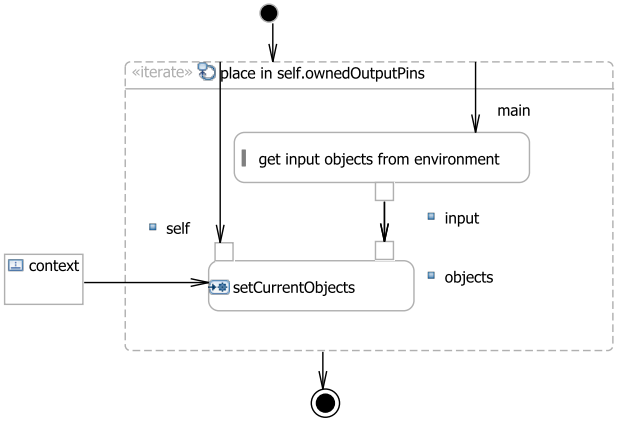


Abbildung 3.23: MInputAction::mInput

Semantik in ASM

Rule MInputAction(thread)
if *thread.currentNode* **is** MInputAction **then**
  forall *p* **in** *InputPlaces(thread.currentNode)* **do par**
    CurrentObjects(top(thread.stack), p) := ReadInput(p)
  enddo
endif

monitored *ReadInput*(*place*)

3.3.8 Atomic Group

Zur Protektion einer Reihe von Aktionen vor Unterbrechung durch parallel ausgeführte Aktionen anderer Threads kann eine *Atomic Group* verwendet werden. Eine definierte Aktionsgruppe ist in ihrem Verhalten atomar, d.h. alle in ihr enthaltenen Aktionen verschmelzen zu einer Transformation des Laufzeitmodells, bei dem keine Zwischenzustände sichtbar werden. Somit aggregieren sich alle Änderungen zu einer beobachtbaren Zustandstransition.

Abstrakte Syntax

AtomicGroup ist als Ableitung der Klasse `MActionGroup` definiert und gruppiert alle über `ownedActions` enthaltenen Aktionen (s. Abb. 3.7). Optional kann durch `lockedObjectExpression` eine OCL-Abfrage angegeben werden, die bei Eintritt ausgewertet wird und das Ergebnisobjekt als Synchronisationskontext nutzt (vgl. <*atomic_expr*> in der Syntax). Dadurch können auf Objektebene gezielt gegenseitige Exklusionen von Aktionen/Threads angegeben werden, die streng genommen die generellen Aktionsausschlüsse im **AR** ergänzen, aber für Implementierungen der allgemeinen Semantik einen Hinweis auf Konflikte geben (s. 3.1.1).

Sollten enthaltene Aktionen InvocationActions sein, so werden rekursiv alle aufgerufenen Aktivitäten und Aktionen als *eine* atomare (d.h. beobachtbare) Aktion ausgeführt. Threads, die dabei gestartet werden, beginnen erst nach Austritt aus der Gruppe ihre Arbeit (ausgenommen `Fork-Join`).

Graphische Notation

Ähnlich der IterateAction notiert man atomare Gruppen durch eine Region mit gestrichelter Umrandung, innerhalb derer sich alle gruppierten Aktionen befinden.

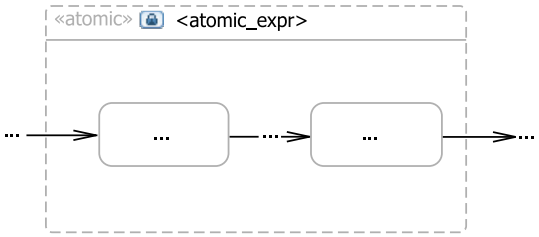


Abbildung 3.24: Atomic Groups

Semantik

Das atomare Zusammenfügen von einzelnen Aktionen verändert die Aktionssemantik eines Threads im Prinzip nicht, sondern beeinflusst primär die Zustandsbildung im Trace. Entscheidender für die Semantik ist der Umstand, dass parallel ablaufende Threads faktisch „warten“, bis alle innerhalb einer AtomicGroup enthaltenen Aktionen abgearbeitet sind. Dies kann theoretisch zu Singularitäten führen, wenn ein Thread z.B. innerhalb einer solchen Gruppe in einer Endlosschleife verharrt: für diesen Prozess gibt es dann eigentlich *keine* nachfolgende Aktion; aus Sicht eines anderen Threads jedoch schon! Da kein Zeitbegriff existiert, könnte man im Sinne des *unbounded nondeterminism* argumentieren, dass *unendlich* viele Schritte möglich sind und die gesamte Ausführung weiter voranschreitet. Ähnlich verhält es sich mit einem unendlichen „fork“ innerhalb der AtomicGroup.

Da diese und weitere Konstruktionen modellierbar sind, stärken sie zwar die theoretische Ausdrucksmächtigkeit der MActions, machen eine solche operationale Semantik aber für keinen Computer ausführbar und haben deshalb für die Praxis keine Relevanz. Für die ASM-Semantik nehmen wir deshalb an, dass die Atomizitätssemantik bereits bei der Thread-Auswahl durch *ResolveConflict* berücksichtigt und aufgelöst wird und verzichten auf eine weitergehende explizite Modellierung (vgl. 2.6.2).

3.3.9 Opaque Action

Als Erweiterungskonzept – zum Beispiel zur Realisierung mathematischer Funktionen, erweiterter Interaktion mit der Umgebung etc. – steht die OpaqueAction zur Verfügung. OpaqueActions haben generell beliebig viele Ein- und Ausgangspins und fügen sich in die allgemeine Aktionssemantik ein, dass heißt sie sind atomar und können verändernd auf das Laufzeitmodell zugreifen.[11] Sämtliche Änderungen müssen Metamodell konform geschehen.

Abstrakte Syntax

OpaqueActions sind durch die Klasse `MOpaqueAction` definiert (s. Abb. 3.7) und haben neben dem `tag` zur Identifizierung des auszuführenden Verhaltens und einem Namen keine weiteren Eigenschaften.

Graphische Notation

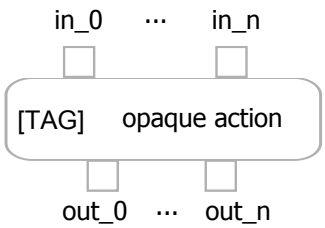


Abbildung 3.25: OpqaueAction mit *Tag*-Kennzeichnung

Semantik

Das eigentliche Verhalten hinter einer OpaqueAction wird über einen *Tag* bestimmt, der in einer MActions Implementierung die zugehörige Aktion ausführt; ihr Verhalten ist implementierungsspezifisch. [12]

[11] OpaqueActions ist insb. nur das Verändern des Laufzeitmodells innerhalb des LZR gestattet, nicht des Zustands der ausführenden Maschine oder des Metamodells.

[12] Hauptmotivation für diese Aktion ist es, eine flexible Schnittstelle zu schaffen, um gängige Simulationsaufgaben wie das Einlesen und Weiterverarbeiten von Daten zu ermöglichen und gleichzeitig die Kernsemantik der Aktionen minimal zu halten. Diese Aktionsform sollte hauptsächlich in Bibliotheken zum Einsatz kommen, die der Erweiterung und Anbindung an die Umgebung dienen.

3.3.10 Hilfsfunktionen des Laufzeitmodells

Zur Strukturierung der Verhaltensbeschreibung wurden einige Hilfsoperationen verwendet, die der Vollständigkeit halber in diesem Abschnitt kurz erläutert werden sollen. Die Initialisierung eines Stapelkontextes erfolgt mittels `init`, dargestellt in Abb. 3.26. Hier wird der initiale Knoten des Verhaltens als erster auszuführender Knoten festgelegt, damit der Ablauf genau dort startet. Weiterhin steht in derselben

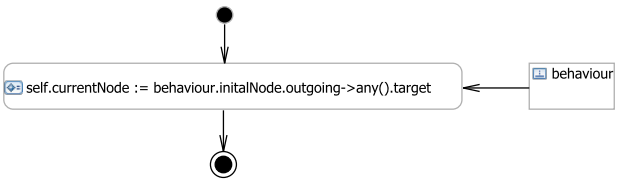


Abbildung 3.26: MContext::init

Klasse `MContext` die Operation `lookupIncarnation` bereit, um in einem Stapelkontext nach Bedarf Inkarnationen für Plätze anzulegen, in denen die Objekte der Plätze abgelegt werden (s. Abb. 3.27). Die Zuordnung geschieht über die logische Instanziierung mittels `metaObject`. Sollte ein Platz bereits in einem Kontext über eine Inkarnation verfügen, wird diese weiterhin genutzt, sonst ein neuer geschaffen.

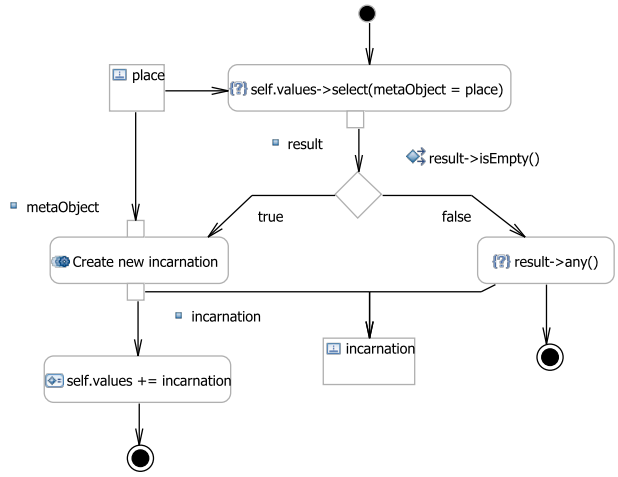


Abbildung 3.27: MContext::lookupIncarnation

Neben den Kontextoperationen ist `MAction::getNextNode` die Hilfsoperation, die das Voranschreiten von einer Aktion zur nächsten festlegt (s.Abb. 3.28). Wie in der Ablaufsemantik weiter oben beschrieben, schreitet der Kontrollfluss zum jeweils nachgelagerten Knoten unter Berücksichtigung des Schlüsselwortes *main* voran.

3.3.11 Reflexion

Aus den Definitionen in diesem Kapitel geht hervor, dass obwohl die MActions metazirkulär beschrieben sind, keine Homoikonizität vorhanden ist. Reflektion beschränkt sich in εMOF auf die Inspektion von Metaebenen (über `metaObject`, `classifier` und OCL-Reflexion), aber sie lässt keine direkte Veränderung oder

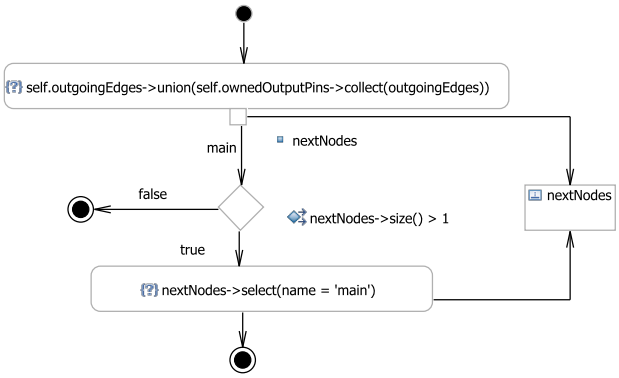


Abbildung 3.28: MAction::getNextNode

Ausführung von Aktionsdefinitionen zu (wie z.B. *eval* in LISP).[13]

3.4 Syntaktische Erweiterungen

Dieser Abschnitt definiert rein syntaktische Konventionen, die die Semantik der MActions nicht erweitern, aber in der Praxis nützliche Abkürzungen darstellen.

Bei der Metamodelldefinition bilden einige Operationen reine Funktionsdefinitionen, die graphisch notiert lediglich aus einer QueryAction bestehen, bei der alle Eingabeparameter genutzt werden um einen Ausgabewert zu berechnen. In Anlehnung an zusätzliche Attribute/Operationen in OCL definieren wir die textuelle Kurzform (vgl. [17], Abschn. 7.4.4.):

context Expression:
def someFunction(**Integer** a, **Integer** b) : **Integer** = (a ∗ a) + (b ∗ b)
  redefines BaseClass#func(**Integer**, **Integer**) : **Integer**

Neben der normalen OCL-Funktionsdefinition besteht die Möglichkeit, über das Schlüsselwort **redefine** die `redefinedBehaviour`-Referenz zu Operationen in Basisklassen mittels deren Signatur anzugeben. Dabei können Ein- und Ausgabetypen in gewohnter Form ko-/kontravariant verändert werden (vgl. Abschn. 3.2.4). Da Abfragen in vielen Fällen direkt zur Belegung eines Platzes an einer nachgelagerten Aktion dienen, erlauben wir ferner die direkte Angabe von OCL-Abfragen an Plätzen und Transitionen (vgl. Abb. 3.29, ein Anwendungsbeispiel s.a. 3.5).

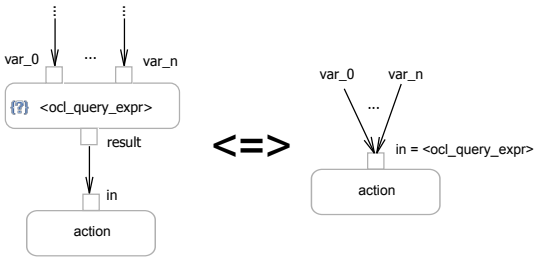


Abbildung 3.29: Kurzform für direkte Abfragen

[13] Sollen Sprachen mit Homoikonizität operational definiert werden, lässt sich dies durch Definition eines Interpreters erreichen, ähnlich frei verfügbarer LISP-Interpreter geschrieben in OO-Sprachen

3.5 Berechenbarkeit

Das MActions intuitiv turingvollständig sind, da die wesentlichen Konstrukte einer WHILE-Sprache enthalten sind, wollen wir im Folgenden beweisen. Die Besonderheiten der Aktionssemantik im Hinblick auf Endlichkeit und Implementierung des Kalküls sind in Kapitel 5.1.1 diskutiert.

Satz 1 *Die Aktionssemantik* MActions *ist turing-vollständig.*

Beweis: Wir beweisen Satz 1 direkt durch Konstruktion einer Turing-Maschine mittels MActions. Auf die Rückrichtung des Beweises wird verzichtet. Eine Turingmaschine (TM) gegeben als 7-Tupel $TM = (Z, z_0, \Sigma, \Gamma, \delta, E)$ mit

1. Z endliche Zustandsmenge,
2. $z_0 \in Z$ dem Startzustand,
3. Σ dem Eingabealphabet,
4. $\Gamma \supset \Sigma$ Arbeitsalphabet,
5. $\delta : Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ Überführungsfunktion
6. $E \subseteq Z$ Endzustände,
7. $\square \in \Gamma - \Sigma$ Blank-Symbol,

kann durch das in Abbildung 3.30 und 3.31 gezeigte MActions-Modell simuliert werden. Die Klasse `TM` erhält ihr Band über die Zeichensequenz `tape` mit `pos` als Lesezeigerposition, sowie `state` als Kennzeichnung der aktuellen Konfiguration. Die Transitionsfunktion δ bildet eine Menge von `DeltaDef`-Objekten direkt ab. Gestartet wird die Maschine mittels dem in Abb. 3.31(b) gezeigten Einstiegspunkt, der mittels der nicht weiter spezifizierten Operation `init` alle Attribute initialisiert und anschließend die Ausführung initiiert.

□

Anmerkung: Durch die Klasse `TM` verschmilzt abstrakte Syntax und Laufzeitmodell der Turingmaschine der Einfachheit halber zu einem Objekttyp. Mittels der nicht weiter spezifizierten Operation `init` wird lediglich direkt ein TM-Laufzeitmodell erzeugt. Des Weiteren wird angenommen, dass diese Operation ein genügend langes Schreib-/Leseband `tape` erzeugt und den Lesekopf mit `pos` an eine *geeignete* Startposition setzt, so dass es während der Ausführung nicht zu einem *undefined* durch den indexbasierten Zugriff kommt.[14]

[14] Alternativ könnte man die Positionierung im Hinblick auf das Band vor jeder Bewegung nach links/rechts prüfen und ggf. durch Kopieren das Band „verlängern“. Für eine Diskussion zur generellen Problematik von Modellierung vs. Implementierung s.a. Kap. 5.1.1

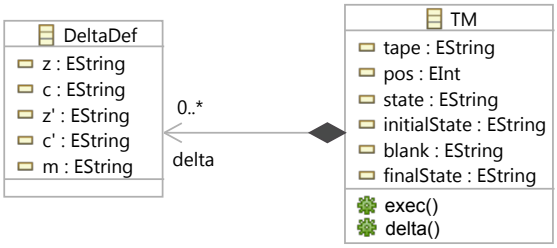
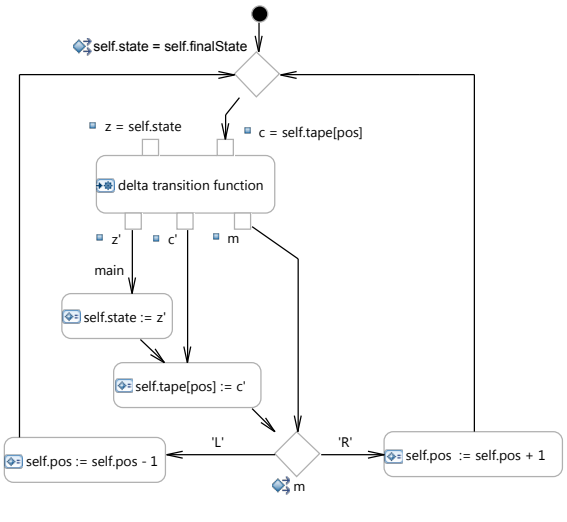
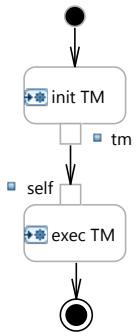


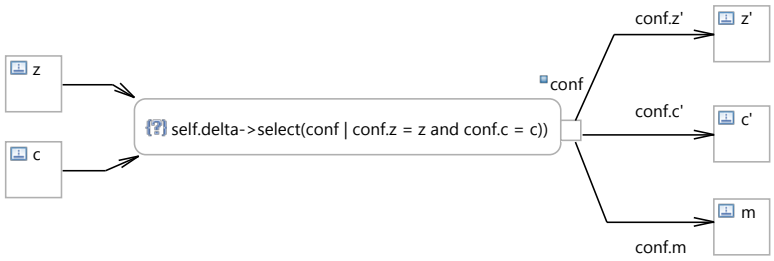
Abbildung 3.30: Turingmaschine: Eine `TM` enthält eine Menge von Transitionselementen `DeltaDef`



(a) Hauptausführungsschleife (`exec`)



(b) Initialisierung



(c) `delta`-Transitionsfunktion

Abbildung 3.31: MActions-Definition der Turingmaschine

KAPITEL 4

Dynamische Modellanalyse

Dynamische Analyse für die Modellsimulation fußt auf der Beobachtung, Messung und Auswertung von Zustandsgrößen und deren Entwicklung über Ausführungsläufe. Das beobachtbare Verhalten eines Modells ist hierbei gebunden an die Zustandsänderungen des Simulationsmodells. Für die weiteren Betrachtungen wird im Folgenden die Herangehensweise im Kontext der Modellsimulation mit MActions eingeordnet, um anschließend die dynamische Modellanalyse metamodellunabhängig, aber basierend auf einer konkreten Aktionssemantik, zu ermöglichen. Zu diesem Zweck werden die bei einer Ausführung entstehenden Modellzustände als Trace aufgezeichnet und im Hinblick auf eine Auswertung klassifiziert. Darauf aufbauend können Konzepte des Verhaltensvergleichs (Bisimulation) sowie der automatischen Verifikation mittels einer prädikatenlogischen Temporallogik (LT-OCL) beschrieben werden, die letztlich nur indirekt (über Zustände) an die operationale Semantik gekoppelt sind. Der Ansatz zeigt auf, wie sich über den gegeben Definitionen eine Klasse von Sprachen zur automatisierten Auswertung definieren lässt und untersucht die sich daraus ergebenden Probleme an zwei Fallstudien.

4.1 Modellsimulation

Basierend auf einem gebildeten Modell besteht ein Simulationsexperiment im Allgemeinen aus den Phasen Initialisierung, Modellausführung, ggf. Beobachtung während laufender Simulation, Aufzeichnung von relevanten Daten, Beendigung der Ausführung und anschließende Aus- und Bewertung. Zusammengefasst können für die Phasen in unserem Kontext die folgenden Modelle/Konzepte eingesetzt werden:

1. *Modellaufbau (Simulationsinitialisierung)*: Die Initialisierung lässt sich im Rahmen eines MOF-Frameworks durch Editoren zur Modelldefinition realisieren und wird im Folgenden als gegeben betrachtet. Es sei an dieser Stelle lediglich angemerkt, das vielfach Editoren anhand von Metamodelldefinitionen abgeleitet werden können. Beispiele hierzu sind u.a. die Frameworks XText [67], TEF [113] für textuelle Definitionen oder GMF [114], Obeo Designer [115] u.a. für graphische Notationen. Für diese Arbeit wurde in den meisten Fällen lediglich ein einfacher, von EMF generierter Standard-Baumeditor verwendet. Die dadurch erzeugten Modelle dienten als direkte Eingabe für eine MActions-Modellausführung.

2. *Modellausführung, Beobachtungen*: Die Modellausführung stellt das Interpretieren von εMOF Modellen mit MActions dar. Die dabei simulierten Modelle können durch Mittel des Frameworks aufgezeichnet, beobachtet und ausgewertet werden. Die Details zur Definition der Aktionen wurden bereits in Kapitel 3 ausführlich erläutert. Neben der interaktiven Steuerung durch Nutzer, steht üblicherweise als erster Schritt eine Validierung des Modells an. Zu diesem Zweck kommen neben der Plausibilisierung und/oder dem Datenabgleich generell auch Techniken zum Einsatz, die wir in Kapitel 4.1.1 einordnen wollen.
3. *Aufzeichnen von Daten*: Neben der Modelldefinition als Initialisierung betreffen weitere Anforderungen an eine Simulationsumgebung Funktionalitäten zum gezielten Anzeigen und Sammeln von Daten während der Simulation. Wir werden in Abschnitt 4.1.1 einige Techniken in Relation zu den MActions einordnen, dann aber mit dem Ziel der dynamischen Analyse detailliert das Aufzeichnen von Ausführungsläufen in Abschnitt 4.1.3 fortfolgend untersuchen.
4. *Auswertung*: Einhergehend mit dem Aufzeichnen von Daten hängen unmittelbar Techniken der Auswertung zusammen. Dazu muss zunächst die Simulation beendet werden, entweder durch einen finalen Zustand oder in Fällen der Endlossimulation durch ein explizites Anhalten des Simulators. In beiden Fällen entstehen Datenmengen (z.B. Szenarien, Traces), deren Analyse man automatisiert angehen will.

Die Aspekte des Aufzeichnens und der Auswertung von Daten stehen für die dynamische Analyse im Vordergrund. Neben informellen und manuellen Methoden der Analyse wollen wir im Folgenden zwei Hauptaspekte betrachten:

1. Aussagen über das ausgeführte bzw. *beobachtbare Verhalten* eines Modells. Dabei ist von besonderem Interesse, wie zwei Modelle aufgrund einer operationalen Semantik mit MActions verglichen werden können. Dieser Fragestellung widmet sich Kapitel 4.2.
2. Automatisierter Nachweis von *dynamischen Eigenschaften*. Prinzipiell können Simulationsdaten durch diverse Algorithmen analysiert werden (z.B. Filter, Sortierung, Transformationen). Uns interessiert in diesem Sinne die direkte Überprüfung mittels einer Temporallogik (vgl. 2.7) und wie diese für MActions-Semantiken generisch funktionieren kann. Dieses stellt Kapitel 4.3 dar.

Grundlage für beide Themenkomplexe bildet die Transformation des Laufzeitmodells während der Ausführung durch MActions-Definitionen und die dabei durchlaufenen Zustände, die wir zunächst genauer beleuchten müssen.

4.1.1 Modellvalidierung, Modellverifikation

Neben der Auswertung und formalen Analyse von Simulationsergebnissen kommt der Visualisierung und Animation eine besondere Rolle zu. Oftmals möchte man dem Simulationslauf durch visuelles Feedback folgen oder diesen u.U. gezielt mit Eingaben steuern. Wie in Kapitel 1 beschrieben bilden Simulationsumgebungen einen Spezialfall von Entwicklungsumgebungen einer (Modellierungs-)Sprache, d.h. es bedarf neben eines Editors für die Simulationsmodelle ebenfalls zusätzlicher Komponenten z.B. für das Modelldebugging, um fehlerhafte Modelle der eigentlichen Intention anzupassen. Diese und weitere Techniken zielen generell entweder auf die *Modellvalidierung* oder die *Modellverifikation*. Die besondere Bedeutung, die dem Laufzeitmodell zukommt, wird anhand der folgenden Liste von Techniken deutlich:

Animation Animationen und Ausgaben können die im Laufzeitmodell definierten Objekte und Attribute visualisieren. Zum Beispiel kann das Laufzeitmodell eines Zustandsautomaten, das spezifiziert, welche Zustände gerade aktiv sind, oder welche Transitionen feuern, ausgewertet und durch Marker/Einfärben der Elemente im Editor (der M1-Modelle) angezeigt werden (s.a. Abschnitt 4.4.1). Weitere Beispiele reichen von virtuellen Oszilloskopen, die reelle Variablenwerte darstellen (z.B. in MATLAB/Simulink) bis hin zu 3D-Animationen und Strömungsfilmen. Faktisch speisen sich die so synthetisierten und visualisierten Informationen fast ausschließlich aus dem Laufzeitmodell.

Modell-Debugger Für die meisten Sprachen ergibt eine schrittweise Abarbeitung in Form eines speziellen *Debugmodus* Sinn, bei dem die Ausführung jeweils nach einzelnen, nutzergesteuerten Schritten wieder anhält und Informationen über den Laufzeitzustand anzeigt. Dabei muss zunächst spezifiziert werden, welche MActions-Sequenzen einem Schritt zugeordnet werden kann. Vorstellbar ist, dass Informationen über *Steps*, *Breakpoints* etc. an die Metamodelle annotiert werden, z.B. in Form von Markern an einer `MOperation`.[1] Des Weiteren sind Debuginformationen letztlich wiederum Darstellungen des Laufzeitmodells, ggf. mittels einer Abbildung auf ein separates *Debuggingmodell* (wie z.B. im *eclipse debugging framework*).

Monitoring Neben der Verfolgbarkeit durch Visualisierungen des Ablaufs ist vielfach eine Ausführungsüberwachung (*Monitoring*, Programmüberwachung) wünschenswert, um (semi-)automatisiert bestimmte dynamische Eigenschaften zu erkennen und ggf. darauf zu reagieren. Zum Beispiel möchte man bei länger laufenden Simulationen Überschreitungen von Grenzwerten erkennen oder im Fehlerfall den Lauf anhalten. Auch hier dient das Laufzeitmodell als Basis für Überprüfungen zur Laufzeit. Als Spezialfall kann hier das Testen mit Zusicherungen (*Assertions*) gesehen werden, das an vordefinierten Interaktionspunkten das „*Model Under Test*“ auf gültige Werte oder Zustände prüft.[2]

Verifikation Neben den *während* der laufenden Simulation stattfinden Überwachungen wollen wir die *nachträglichen* Verifikationen nach abgeschlossenem (oder vorzeitig beendetem) Lauf unterscheiden. Für erstere haben sich auch Begriffe wie „Online-Verifikation“ herausgebildet, gegenüber der „Offline-Verifikation“, bei der man insb. im Bereich der Laufzeitverifikation (engl. *Runtime Verification*) von Programmen und Systemen spricht (z.B. [68][62]). Ansätze zur Online-Verifikation kommen größtenteils ohne das Aufzeichnen von Programmzuständen aus, da zu festgelegten Zeitpunkten eine Analyse der Laufzeitgrößen stattfindet. Verfahren zur nachträglichen Auswertung benötigen mindestens eine Aufzeichnung von allen relevanten Laufzeitdaten, so dass Rückschlüsse auf die Ausführung gezogen werden können. Abhängig von der Zielsetzung reicht u.U. das Aufzeichnen von einem Teil der Zustandsgrößen (z.B. Zeilennummern für eine einfache Code-Coverage-Analyse).

Diese Liste ließe sich noch weiter fortsetzen und detaillieren. Um für die genannten Funktionalitäten adäquate Werkzeugunterstützung *automatisiert* zu erhalten, müssen in der Regel zusätzliche Annotationen und Ergänzungen im Metamodell vorgenommen werden (z.B. für die Syntax [114][67]). Sofern dies generisch möglich ist, werden in der Praxis vielfach Einzelfalllösungen nötig, um wirklich effiziente

[1] Komplexere Debugoperatoren wie z.B. *step into* oder *step return* bei OO-Programmierumgebungen bedürfen sicherlich separaten Abbildungsmodellen. Für eine beispielhafte Umsetzung einer *single step*-Semantik s.a. Blunk [116]

[2] Ein „echtes“ Beispiel für Tests ist ein *back-to-back*-Test von Simulation und Implementierung

Entwicklungsumgebungen für eine Sprache zu erhalten. Ohne auf diesen Themenkomplex und der damit verbundenen Problematik des Generierens von Werkzeugen aus Metamodelldefinitionen einzugehen,[3] können wir feststellen, dass den angesprochenen Methoden letztlich gemein ist, die operationale Semantik und das Laufzeitmodell als Basis zu nutzen, um Erkenntnisse aus dem Modellverhalten zu ziehen.

Im Folgenden wollen wir genauer auf die Möglichkeiten zur dynamischen Verifikation anhand von aufgezeichneten Zustandsdaten eingehen. Dazu beschränken wir uns auf den Fall, sämtliche Änderungen aufzuzeichnen die notwendig sind, um *jeden durchlaufenen* Zustand der Simulation nachträglich *vollständig* zu rekonstruieren. Wir argumentieren, dass dieser Ansatz für alle anderen denkbaren Analysen, denen ein Teil der Informationen genügen würde, ebenfalls funktioniert (auch wenn man aus Gründen wie Datenvolumen/Performance üblicherweise in der Praxis gezielter Daten sammeln würde). Als Voraussetzung muss zunächst ein genauer Blick auf die Zustände geworfen werden, die bei einer MActions-Modellausführung beim Laufzeitmodell entstehen.

4.1.2 Modellzustände

Das beobachtbare Verhalten einer Simulation ist eng gekoppelt an die Zustandsgrößen des sich verändernden Laufzeitmodells, welches sich auf mindestens drei Ebenen äußert:

(1) Ausgaben des ablaufenden Modells an seine Umgebung. Der offensichtlichste „Zustand“ und somit das beobachtbare Verhalten eines Modells äußert sich durch explizite Ausgaben an die Ausführungsumgebung. Hierzu zählen z.B. Werte von Variablen, erzeugte textuelle und/oder graphische Ausgabedaten. Diese können von der Umgebung aufgezeichnet werden und repräsentieren den *extern (sichtbaren) Zustand* des Modells.

(2) Änderung von modellinternen Zustandsgrößen. Hierunter zählen alle internen Variablen, Objekte und Strukturen die durch das ausführende Modell angelegt und verändert werden. Hinzugerechnet werden auch etwaige Verwaltungsstrukturen, Programmzähler, Registerwerte etc. Bei einem Java-Programm sind das z.B. alle Objekte mit Feldern und Wertebelegung, lokale Variablen, erzeugte Threads mit akquirierten Monitoren, Stacks sowie geladene Klassen. Deshalb bezeichnen wir diesen Zustand als den *internen Zustand eines Modells* und dieser entspricht genau den Objekten des Laufzeitmodells.

(3) Änderungen der Laufzeitumgebung. Bezieht man die Ausführungsumgebung mit ein, ergeben sich unter Umständen „Zwischenzustände“, die der Überführung eines internen Modellzustandes in einen anderen dienen. Zum Beispiel muss bei einer Programmiersprache für einen Funktionsaufruf in der Regel ein neuer Stackkontext hergestellt, Parameterwerte oder -referenzen kopiert und der Programmzähler weiter gesetzt werden. Für das Beispiel Java wären dies einzelne Schritte im Bytecode, aber auch sog. ”micro languages” von Prozessoren fallen in diese Kategorie. Für Sprachdefinitionen mittels einer operationalen Semantik wie der MActions gilt gleiches: die Metamodelldefinitionen beschreiben detailliert, welche Objekte und Attribute wie manipuliert werden, jedoch nicht *wann* ein gültiger interner Zustand erreicht wird. Wird eine Sprache also im Allgemeinen mittels einer Metasprache für die operationale Semantik beschrieben, so sind die Zustände der Metasprache als *Mikrozustände* zu trennen vom internen Zustand des Modells.

[3] Für eine weitergehende Analyse s.a. [71]

Es ist anzumerken, dass weitere *ausgezeichnete* Zustände innerhalb eines Modells in unserer Betrachtung eine Rolle spielen werden. Damit sind Zustände gemeint, die nicht auf der reinen Sprachbeschreibungsebene vorkommen, sondern während der Ausführung von Modellen. Diese wollen wir als *Makrozustände* bezeichnen, da sie (ohne die Ein-/Ausgaben an die Umgebung zu betrachten) logisch auf der „Zustandsskala“ das Ende bilden (in diesem Bild bilden Mikrozustände den Anfang). Zum Beispiel kann bei der Programmiersprache C ein Programm selbst einen Zustandsautomaten beschreiben (z.B. über eine Variable und eine `switch` Anweisung), der dann in der Analyse eine besondere Rolle spielt. Ein anderes Beispiel sind Prä- und Postkonditionen von einzelnen Methoden, bei denen man vor bzw. nach dem Aufruf Prädikate oder Invarianten prüfen möchte. Auf ausgezeichnete Zustände dieser Art werden wir in exemplarischen Analysen zurückkommen (vgl. Abschnitt 4.4.3, sowie A.4).

Betrachten wir die Relation zu MActions, so sollte intuitiv klar sein, dass Ausgaben nur durch MOutput-Aktionen getätigt werden können. Somit erhalten wir alle Zustände der Ebene (1) durch Observierung dieser Aktionen. Die Terminologie *externer Zustand* wurde gewählt, um die Relation eines externen Beobachters im Augenblick der Observation (einer Ausgabe) zum gerade existierenden (internen) Zustand des Modells herzustellen.[4]

Die Trennung zwischen Ebene (2) und (3) mag auf den ersten Blick verwirren und irrelevant erscheinen, ist jedoch in der Praxis von höchster Relevanz.[5] Während andere Kalküle wie ASMs oder SOS oftmals direkt auf eine Beschreibung des internen Modellzustandes abzielen, drücken MActions zunächst einmal nur Mikrozustände und deren Übergänge aus. Im Folgenden werden wir zeigen, wie sich mittels gezielter Auswertung der MActions alle Ebenen zur Analyse unterstützen lassen. Eine weitergehende Diskussion zur Zustandsunterscheidung und Einordnung findet in Kapitel 5.2.1 statt.

4.1.3 Aufzeichnen von Abläufen

Das Aufzeichnen von Ausführungsläufen ist der erste Schritt zur nachträglichen Auswertung einer Modellausführung. Aufgezeichnete Abläufe manifestieren sich als sog. *(Execution) Traces*[6]. Der Zustand des Modells muss dabei so konserviert werden, dass alle für die Analyse relevanten Informationen eines Ausführungslaufs zur Auswertung *nachträglich* zur Verfügung stehen bzw. rekonstruiert werden können. Das hier entwickelte Tracemodell stellt im Gegensatz zu anderen Ansätzen der Programm- und Simulationsauswertung eine generische Datenstruktur zur Verfügung, die Modellabläufe erfassen kann und bietet nicht nur eine Schnittstelle zum Zugriff auf Laufzeitdaten an, wie bei Simulationsumgebungen üblich. Aus den bisherigen Definitionen sollte offensichtlich werden, dass es genügt, dazu das Laufzeitmodell aufzuzeichnen, da hier alle Zustandsgrößen enthalten sind (sowohl Mikrozustände als auch interne).

Mit den Definitionen aus 2.5 ließe sich ein Trace mathematisch als die geordnete Menge aller Zustände des Laufzeitmodells beschreiben. Konzeptionell werden

[4] Dies ist äquivalent zum Beispiel zu den *Observations* von Bloom für CCS, wenn wir uns auf Ausgaben beschränken (vgl. [96]). Bloom betrachtet allerdings eine abstrakte Menge von Observationen für die Definition von Kongruenz/Bisimulation (s.a. 4.2.1)

[5] Insb. bei Sicherheitsanalysen von Programmen im Automobilbereich, Flugzeugbau, Eisenbahn- und Verkehrswesen spielt die Betrachtung der RTL/ALU-Adressierungsebene eine besondere Rolle bei der Identifikation von Hardwarefehlern

[6] die deutschen Begriffe 'Spur' oder 'Pfad' haben sich hier nicht durchgesetzt und wir verwenden deshalb den englischen

wir diesen Weg auch beschreiten, geben allerdings zunächst ein generisches *Trace-(Meta-)Modell* an, welches Zustandsänderungen an Modellen als Delta der Zustände erfasst. Anschließend binden wir Traceinstanzen an den algebraisch formaleren Zustandsbegriff.

Definition 26 *Ein* Trace *ist definiert als Instanz der Klasse* `Trace` *des in Abb. 4.1 dargestellten Metamodells. Er wird erzeugt durch die* MActions *Ausführungsumgebung während der Modellsimulation und erhält die folgende Struktur:*

1. *Für jede Instanz der Klasse* `MThread` *des* MActions*-Laufzeitmodells wird eine korrespondierende Instanz der Klasse* `Trace` *angelegt. Jede Instanz erhält eine global eineindeutige* `id`*, die der Thread-ID entspricht.*
2. *Jede Änderung an einem Objekt des Laufzeitmodells innerhalb eines Threads erzeugt einen* `ObjectChange` *Eintrag, der die Änderung im Detail beschreibt. Diese Einträge werden mittels* `Change`*-Instanzen gruppiert, welche wiederum pro Stackkontext (d.h.* `MContext`*) zusammengefasst werden (vgl. auch 3.2.5). Jede dieser Instanzen enthält den Satz von Änderungen, die durch alle Aktionen im selben Kontext erzeugt worden sind. Das Attribut* `actionURI` *hält zusätzlich eine Referenz auf die* `MActivity`*-Definition, welche zu dem Stackkontext ausgeführt wurde. Die Beziehungen* `changes` *und* `nestedChanges` *dienen zur Abbildung des Aufrufstacks.*
3. `ObjectChange` *bezeichnet das veränderte Objekt mittels eineindeutiger* `objectID` *und Referenz auf die Metaklasse* `metaClassName` *als URI. Es wird für den aktuellen Kontext jeweils in die* `objectChanges` *Liste eingetragen. Als Sonderfall der Objekterzeugungen ist diese Liste leer.*
4. *Instanzen von* `FeatureChange`*, enthalten in der Liste* `featureChanges` *einer Objektänderung, beschreiben welche Eigenschaften sich wie geändert hat:*
   (a) `featureName`*: Referenz auf die Objekteigenschaft in der Metaklasse des Objekts.*
   (b) `valueAsString`*: enthält den gesetzten (neuen) Wert als String. Dieser ist wie folgt aufgebaut:*
      i. *Attribute mit primitivem Typ oder* `EDataType` *liegen als XMI/XML Wert vor, d.h. z.B. für Attribute vom Typ* `EBoolean` *entweder true oder $false$, für* `EDouble` *die Fließpunktnotation der IEEE 754-1985 etc. (vgl. [78]). Bei mehrwertigen Attributen sind die Werte durch Leerzeichen voneinander getrennt.*
      ii. *Bei Referenzen zu anderen Objekten werden eineindeutige Objekt-IDs serialisiert, abhängig von der Multiplizität des Attributs ggf. durch Leerzeichen getrennt.*
5. *Durch die Vererbung von* `AbstractChange` *wird zusätzlich an allen Instanzen ein Zeitstempel* `instant` *verzeichnet, der den logisch-kausalen Zeitpunkt der Änderung angibt (vgl. 3.1.2). Dieser kann für gleichzeitig ablaufende Aktionen an verschiedenen Instanzen denselben Wert enthalten.*

Im Folgenden sollen einige Hintergründe und Konsequenzen der Definition 26 erläutert werden, bevor der Zustandsbegriff für Modelle eingeführt wird. Zunächst sollte angemerkt werden, dass die gewählte Struktur des Tracemodells willkürlich ist, jedoch die geforderte Rekonstruierbarkeit aller Modellzustände durch das sequentielle Aufzeichnen der Zustandsdifferenzen erfüllt. Diese lassen sich durch sukzessive Anwendung der aufgezeichneten Differenzen auf das initiale Laufzeitmodell ableiten. Damit erfüllt das Tracemodell im Kern die Mindestanforderung durch die Klassen

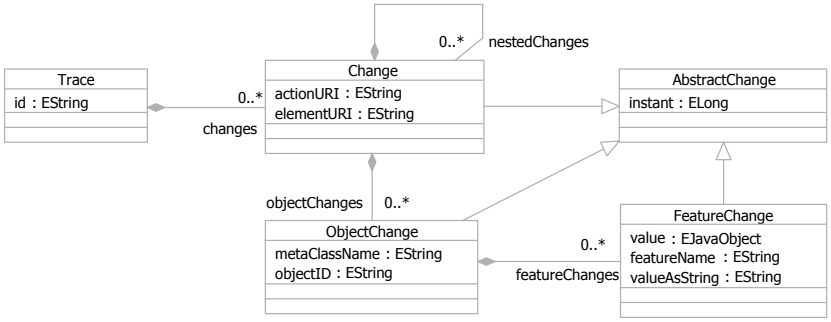


Abbildung 4.1: Generisches Trace-Metamodell über Modelländerungen

`ObjectChange` und `FeatureChange`. Die Gruppierung durch Instanzen der Klasse `Change` helfen lediglich bei der Auswertung, da alle Änderungen einer `MActivity` unter einem Objekt mit eindeutiger Referenz auf die Aktivität zusammengefasst werden. Dadurch bildet sich die Stapelaufrufstruktur der Laufzeitumgebung als Hierarchie im Trace ab. Über Konsequenzen dieser Struktur im Hinblick auf eine Auswertung siehe Abschnitt 4.2.1.

Des Weiteren bilden mehrere Traceinstanzen die Laufzeitgegebenheiten unmittelbar ab und lassen eine Zuordnung der parallel ausgeführten Aktionen zu Threads zu. In Kombination mit den Zeitpunkten der Änderungen lassen sich die zeitlichen Abläufe sowohl rekonstruieren und in eine (Halb-)Ordnung bringen, als auch während der Aufzeichnung geschickt manipulieren, um *echte* Gleichzeitigkeit zu simulieren. Dabei kann z.B. eine Simulationsumgebung zwei Änderungen als „gleichzeitig abgelaufen“ markieren, indem der Wert des `instant` Attributs beim Aufzeichnen den selben Wert erhält, obwohl ggf. eingeschränkte Simulationsumgebungen (z.B. wegen Prozessoren, Hardware) keine echte Parallelität ermöglichen.

Wie eingangs erwähnt, gibt es wenig generische Modelle, die objektorientierte Strukturen abbilden. Neben dem *Ecore Change Model*, das als Basis für Undo-/Redo bei Operationen über EMF-Modellen dient und nicht primär zum Zwecke des Aufzeichnens von Abläufen erstellt wurde (s. [72]), gibt es eine Reihe von Ansätzen die auf konkreten Programmiersprachen basieren (z.B. [117]). Der Hauptzweck ist nicht die Visualisierung der Objektinteraktionen, sondern ein exaktes Festhalten aller Systemzustände (ohne in puncto Datenhaltung minimal zu sein).

4.1.4 Analyse von Modellzuständen

Die in Kapitel 2.5.4 gegebenen Definitionen zur Aktionssemantik und Traces sind auf einen allgemeinen Zustandsraum gerichtet. In diesem Abschnitt führen wir nun den Begriff für MOF-Laufzeitmodelle ein, die uns sowohl zum Anschluss und Vergleich der „zustandsorientierten Sicht“ bei Simulation und Modellanalyse dient, als auch zur weiteren Definition der Analysesprache LT-OCL.

Definition 27 (Induzierter Zustand) *Sei $\pi = \gamma_0, \ldots, \gamma_n (n \in \mathbb{N})$ eine global geordnete Sequenz von Instanzen der Klasse `Change` aufgezeichnet als Menge von `Traces`, wobei n den Wert des Attributs instant angibt. Dann stellt das Modell $\sigma_i(\gamma_i)$ mit $i \leq n$ einen* induzierten Zustand *von γ_i dar, der sich durch sukzessive Anwendung aller Änderungen $\gamma_0 ... \gamma_i$ auf das initiale Laufzeitmodell ergibt. Synonym bezeichnen wir das somit konstruierte Laufzeitmodell eines indizierten Zustands als den* „Schnappschuss“ *zum Zeitpunkt i.*

Die gesamte Länge eines Traces π bezeichnen wir ferner mit $|\pi|$, womit der Endzustand des Laufzeitmodells also durch $\sigma_{|\pi|}(\gamma_{|\pi|})$ gegeben ist. Aus der Definition geht unmittelbar hervor, dass induzierte Modellzustände zunächst jeden Mikrozustand beschreiben können. Demnach resultiert per se jedes Update einer ausgeführten Aktion in einem neuen Zustand $\sigma(\gamma)$, wobei wir explizit nicht fordern, dass alle γ_n von der gleichen Thread Instanz kommen. Anders gesagt stellt die vollständige Sequenz einer Ausführung $\sigma_i(\gamma_i)$ eine Projektion des Laufzeitverhaltens als lineare Teilordnung dar (Linearisierung aller Änderungen).

4.1.5 Von Mikrozuständen zu internen Zuständen

Um die oben angesprochene Unterscheidung zwischen Mikrozuständen und internen Modellzuständen auch in der operationalen Semantik zu trennen und frei modellierbar zu machen, führen wir neue Konzepte ins MActions-Metamodell ein. Diese dienen ausschließlich der Kontrolle des internen Modellzustands, unabhängig von der Update-Semantik einzelner Aktionen und spezifizieren, wann einzelne Change-Objekte im Trace erzeugt werden. Diese Art der *Aggregation* von (unerwünschten) Zwischenzuständen gibt einem Sprachentwickler Kontrolle darüber, wie sich Zustandsübergänge für die dynamische Analyse darstellen sollen.

Im *kanonischen Modus* der Modellausführung erzeugt jede ausgeführte Aktion mit Update-Semantik einen einzelnen `ObjectChange` im Trace. Implizit wird somit durch jede i-te Aktion ein neuer Zustand $\sigma_i(\gamma_i)$ erzeugt. Im *aggregierten Modus* werden Zustandsänderungen zunächst von der Ausführungsumgebung gesammelt und gezielt durch folgende Konzepte in einen Trace geschrieben:

1. *State Generating Transitions (SGTs)* ermöglichen ein gezieltes Erzeugen eines neuen Zustands. Dazu werden alle bis dato gesammelten `Change`-Objekte als ein einzelner Eintrag (mit beliebig darin enthaltenen `ObjectChanges` und `FeatureChanges`) im Trace hinterlegt. Mit anderen Worten im aggregierten Modus wird bei fehlenden SGTs nie ein Zustand erreicht!
2. *Atomic Groups* kombinieren mehrere einzelne atomare Aktionen in eine neue, unteilbare Einheit. Diese wird während der Ausführung nicht unterbrochen und erzeugt im kanonischen Modus ein einziges `Change`-Objekt. Im aggregierten Modus haben SGTs Priorität und erzeugen auch im Kontext einer MAtomicGroup eine neuen Eintrag im Trace.

Für den Sprachdesigner ergibt sich durch diese beiden Konzepte die Möglichkeit, gezielt den internen Zustand einer Sprache orthogonal zur eigentlichen dynamischen Semantik zu spezifizieren. So kann z.B. eine Event-basierte Sprache, die die Kommunikation von Events über Kanäle an Empfänger verteilt exakt angeben, wann ein neuer „echter“ interner Zustand erreicht wird.

4.2 Verhaltensäquivalenz

Basierend auf der formalen Definition einer Sprachsemantik stellt sich für die Programmanalyse und -verifikation die Frage nach der Verhaltensäquivalenz von Programmen. Durch die Arbeiten von Milner zu CCS [88], Hoare zu CSP [89], Bloom, Meyer et al. zur Bisimulation [93], sowie Hennessy, Pneuli, van Glabbeek et al. [118] zu Prozessalgebren und Temporallogiken wurde eine ausgereifte und abstrakte Theorie entwickelt, die unterschiedliche Formen von Verhaltensäquivalenzen unterschei-

det.[7] Für die Analyse von Metamodellen mit MActions interessiert uns der Zusammenhang im Hinblick auf diese Bisimulationstheorie, der in diesem Abschnitt erarbeitet und in Richtung des Verhaltensvergleichs von Metamodellen mit verschiedenen operationalen Semantiken dargestellt wird.

4.2.1 Isomorphie, Bisimulation und Trace-Äquivalenz

Die Grundlage der Vergleiche bilden Prozessbeschreibungssprachen (meist in Form von GSOS-Regeln), die Prozesse mit Aktionen und deren Kommunikation beschreiben (vgl. [96]). LTS, Synchronisationsbäume, Prozessgraphen und -algebren, u.a. bilden Prozessabläufe semantisch ab, d.h. sie stellen das Verhalten eines Prozesses mit möglichen Abläufen dar. Für verzweigte Prozessstrukturen werden dann über Äquivalenzklassen der Graphen verschiedene Formen von Kongruenzrelationen definiert, abhängig davon, ob über den Zeitverlauf der Prozessausführung entstehende Verzweigungen berücksichtigt werden oder nicht. Mit den Definitionen zu LTS aus Kapitel 2.5 besteht immer dann eine *Simulationsrelation* zwischen zwei Prozessen (d.h. zwei LTS) P, Q, geschrieben $P \rightarrow Q$, wenn für jede Aktion in P eine entsprechende Aktion von Q ausgeführt werden kann. D.h. P kann unter Umständen zusätzlich *mehr* (oder andere) Aktionen ausführen als Q, aber mindestens eben die durch P definierten (in genau gleicher Reihenfolge). Eine *Bisimulationsrelation* (geschrieben $P \leftrightarrows Q$) besteht ferner, wenn sowohl $P \rightarrow Q$ als auch $Q \rightarrow P$. Generell unterscheidet man die folgenden Äquivalenzklassen (vgl. [118]):[8]

Isomorphie ($P \equiv Q$): Zwei Prozesse sind isomorph zueinander gdw. sich ihre Prozessgraphen isomorph aufeinander abbilden lassen. Alle Knoten und Kanten (mit allen Verzweigungen) müssen exakt aufeinander abbildbar sein. Isomorphie kann als Identitätskriterium für Prozesse gelten.

Bisimulation ($P \leftrightarrows Q$): Zwei Prozesse sind bisimilar wie oben beschrieben gdw. für jede Aktion von P eine entsprechende Aktion in Q ausgeführt werden kann und umgekehrt (rekursiv für alle Aktionen in P, Q). Diese Abschwächung gegenüber der Isomorphie betrachtet also lediglich das *tatsächliche* Verhalten in puncto Aktionen, unabhängig von den möglichen Pfaden, Konfigurationen etc. Je nach Definition von *Observation* und „versteckten“ (τ-) Aktionen innerhalb der Prozesse, werden weitere Unterkategorien definiert wie z.B. *Strong*- und *Weak*-Bisimulation, *η-Bisimulation*, *Ready Simulation* u.s.w.

Trace-Äquivalenz ($P \equiv_{tr} Q$): Zwei Prozesse sind traceäquivalent in ihrem gesamten Verhalten gdw. jeder Trace von P auch in Q enthalten ist und umgekehrt. Dieses schwächste Kriterium für Verhaltensgleichheit kann durch weitere Kongruenzbedingungen z.B. anhand von Markierungen im Trace aufgewertet werden, z.B. *decorated trace equivalence*, *colored trace equivalence* etc. Somit können — zusätzlich zur Menge der Traces – ebenfalls mögliche Parallelitätseigenschaften für den Vergleich zweier Prozesse herangezogen werden.

Für eine MActions-Spezifikation ergibt sich das Problem, dass die Transitionssemantik nicht über LTS oder GSOS-Regeln definiert ist: der gesamte Zustandsraum mit möglichen Transitionen ist lediglich durch Änderungsaktionen beschrieben und müsste reformalisiert werden, um die aufgelisteten Äquivalenzklassen direkt anzuwenden. Ohne diesen Weg zu gehen können wir die prinzipielle Idee der unterschied-

[7] Genau genommen existiert mittlerweile ein ganzer Satz von sich überschneidende Formalisierungen, auf die wir im Zuge dieser Arbeit nicht in der Gänze eingehen können. So wurden mittlerweile die Arbeiten zu Prozessalgebren und zur Bisimulation z.B. auf stochastische Automaten übertragen.

[8] Für weitere Details zur Unterteilung und Diskussion sei auf Glabbeek et al. verwiesen.

lichen Äquivalenzklassen auch auf εMOF-Modelle übertragen, indem wir die Verhaltensäquivalenz „neu“ interpretieren und auf Aktionssemantiken anwenden. Auch wenn die Argumentation informell erfolgt, zeigen sich für die praktische Anwendbarkeit wichtige Merkmale, die die folgenden Abschnitte erläutern.

4.2.2 Trace-Äquivalenz für εMOF-Modelle, Verhaltensisomorphie

Das grundlegende Problem bei jedem Vergleich zwischen zwei Modellausführungen im εMOF-Kontext ist die zum Vergleich dienende Äquivalenzrelation für *Objekte*. Egal, ob man über Traces und den darin enthaltenen inkrementellen *Änderungen* am Laufzeitmodell oder direkt über Modell*zustände* ansetzt, die Objektidentität in Zusammenhang mit der Referenz- und Wertsemantik von Attributen stellt das Hauptproblem dar, da zwei Abläufe, die Objekte erzeugen und Referenzierungen herstellen, immer zwei unabhängige Objektgraphen erzeugen, die in geeigneter Form auf Isomorphie überprüft werden müssen. Dabei wird z.B. im Trace jede Objektänderung eines Objekts mit id_x im Verlauf immer wieder unter diesem Bezeichner auftauchen, während in einer zweiten Ausführung das äquivalente Objekt unter id_y geführt wird und allein deshalb (zwangsläufig) zu unterschiedlichen Einträgen im Trace führt. Aus dieser Tatsache heraus definieren wir die Objektäquivalenz als „Struktur- und Wertegleichheit“ wie folgt:

Definition 28 (Äquivalenz von Objekten) *Zwei Objekte o_1, o_2 des Instanzmodells (vgl. 2.3.2) sind äquivalent, geschrieben $o_1 \equiv_M o_2$ gdw. die folgenden Bedingungen rekursiv erfüllt sind (Operation equiv in OCL ergibt true):*

1. *Beide Objekte sind Instanzen derselben Metaklasse:*

   ```
   def equiv(o1 : Object, o2 : Object) : Boolean :=
     o1.classifier = o2.classifier and o1.metaObject = o2.metaObject
   ```

2. *Alle Slots haben äquivalente Referenzen und Werte:*

   ```
   o1.slots->forAll(s1 | o2.slots->exists(s2 | equiv(s1.values, s2.values))) and
   o2.slots->forAll(s2 | o1.slots->exists(s1 | equiv(s1.values, s2.values))),
   ```

   wobei für die verschiedenen Typen gilt:

   (a) *String, Integer, Natural, Boolean, Enum:*

   ```
   def equiv(v1 : OclAny, v2 : OclAny) : Boolean := v1 = v2
   ```

   (b) *Collections:*[9]

   ```
   def equiv(c1 : Collection(OclAny), c2 : Collection(OclAny)) : Boolean :=
     c1->size() = c2->size() and c1->forAll(v1 | c2->exists(v2 | equiv(v1, v2))
       and c1->indexOf(v1) = c2->indexOf(v2))
     and c2->forAll(v2 | c1->exists(v1 | equiv(v1, v2))
       and c2->indexOf(v2) = c1->indexOf(v1))
   ```

Man beachte, dass wir für die Objektidentität (Attribut `identifier`) explizit *nicht* Gleichheit fordern, sondern nur „referenzielle“ Gleichheit im Objektgraphen.[10] Vereinfacht gesagt können zwei Modelle als äquivalent betrachtet werden, wenn alle vorhandenen Objektgraphen und Attribute aufeinander abbildbar sind. Dazu müssen

[9] Für die nicht-sortierten Mengen *Bag, Set, MultiSet* ist die Äquivalenz der Indizes mit `indexOf` auszunehmen (hier nur der Verknappung halber in einem OCL-Ausdruck aufgeschrieben)

[10] Eigentlich wurde bislang nicht festgelegt, wie und welche Objekt-IDs vergeben werden. Eine Implementierung könnte für jeden Ablauf dieselbe Menge von IDs verwenden, wie es z.B. in der Java-VM geschieht. Da wir Objekte aber über ihren Lebenszyklus hinaus identifizieren wollen, nehmen wir an, dass dieselbe ID nicht mehrfach auftritt.

Instanzen aller Metaklassen jeweils ein korrespondierendes Objekt im anderen Modell besitzen:

Definition 29 (Isomorphie von Modellen) *Zwei Modelle M_1, M_2 sind isomorph zueinander, wenn für alle Instanzen einer jeden Metaklasse* Class *gilt:* $\text{Class}_{M1}.allInstances() \equiv_M \text{Class}_{M2}.allInstances()$.

In 4.1.3 haben wir für MActions ein Tracemodell definiert, das für jeden ausgeführten Thread eine Traceinstanz anlegt. Mit diesen Definitionen über dem Instanzmodell lässt sich die Äquivalenz metamodellunabhängig festlegen und wiederum selbst auf das Trace-Metamodell aus Abbildung 4.1 anwenden! Damit lässt sich für zwei Threads T_1, T_2 die Trace-Äquivalenz $T_1 \equiv_{tr} T_2$ ableiten, genau dann wenn die erzeugten Traces π_1, π_2 modelläquivalent sind: $\pi_1 \equiv_M \pi_2$. Allerdings müssen wir für Traces die Attribute `id`, `objectID` und `instant` analog zu Definition 28 herausfiltern, da diese dieselben Informationen des Instanzmodells halten. Für eine vollständige Verhaltensäquivalenz von zwei Ausführungen P, Q (mit beliebiger Threadanzahl) können wir die Äquivalenz $P \equiv_{tr} Q$ nach Definition 29 festlegen, wenn für alle $T_P \in \text{MThread}(P)$ genau ein $T_Q \in \text{MThread}(Q)$ existiert, mit $\pi(T_P) \equiv_M \pi(T_Q)$ und umgekehrt. (Mit $\text{MThread}(X)$ seien *alle* durch die MActions-Definition von X erzeugbaren Traces bezeichnet.)

Diese Äquivalenz ist also über Trace-*Modelle* und nicht über den ausgeführten *Aktionen* definiert. Dies ist sinnvoll, da z.B. durch eine QueryAction ein unterschiedliches Resultat bei derselben Abfolge von Aktionen entstehen kann, man aber eigentlich an dem jeweils entstehenden Zustand des Laufzeitmodells interessiert ist. Im Gegensatz zu der eingangs in 4.2.1 beschrieben Trace-Äquivalenz spiegelt also diese Äquivalenz die *Änderungen* an allen im Laufzeitraum erzeugten Objekten wider.

Weiterhin ist anzumerken, dass die so erhaltene Trace-Äquivalenz sehr eng an die gerufenen Aktivitäten angelehnt ist. Zum einen ist dies durch die Hierarchie über `nestedChanges` der Fall, die den Aufrufstapel der Aktionsausführung auf die Tracestruktur abbildet. Eine Möglichkeit dies zu flexibilisieren (und eine schwächere Form der Äquivalenz zu erhalten) bestände darin, die Tracestruktur zu „verflachen“ und alle `Change`-Instanzen direkt in `changes` einzutragen (vgl. Tracemodell 26, Abb. 4.1, d.h. `nestedChanges` zu eliminieren). Nach Bedarf kann an dieser Stelle weiter gefiltert werden. Zum Beispiel könnten die zeitlichen Abstände der Aktionen, die implementierungsspezifisch sind und in `instant` aufgezeichnet werden, beim Vergleich eine Berücksichtigung finden. Zum Beispiel könnte das Framework über OpaqueAction Erweiterungen spezifizieren, die ein „Warten“ auf Zeitgebern realisieren. Daraus resultierende Zeitschranken im Trace könnten bei der Äquivalenz (u.U. mit einem Δt als Toleranzintervall) berücksichtigt werden.

In Relation zu den eingangs beschriebenen Vergleichskategorien benötigen wir für die strengste Form von Verhaltensäquivalenz, der Isomorphie, weitere Zusammenhänge zwischen den einzelnen Traces, d.h. wann welcher Thread aus einem anderen hervorgegangen ist. Da das Tracemodell den tatsächlichen Ablauf der Aktionen eines Threads mittels `actionURI` bereits mit einbezieht, fehlt für den Zusammenhang von zwei Traceinstanzen eigentlich nur eine Referenz zwischen den ausführenden Threads. Wenn aufgezeichnet würde, welcher Thread durch eine InvokeAction (mit `startThread`) welchen anderen Thread initiiert, und dieses als Referenz zwischen Traces der beteiligten Threads aufgezeichnet würde, könnte Isomorphie so definiert werden, dass der nun zusammenhängende Threadgraph als modelläquivalent gelten muss.

4.2.3 Bisimulation als Äquivalenzrelation der Modellebene

Bisimulation, übertragen auf Zustandsänderungen am Laufzeitmodell, lässt sich ähnlich der Trace-Äquivalenz über Modellisomorphie definieren (vgl. Def. 29). Während in Abschnitt 4.2.2 die Menge der Traceinstanzen mit allen Objektänderungen zum Vergleich dienten, sollen nun die Zustände des Laufzeitmodells zu *bestimmten Zeitpunkten* verglichen werden, unabhängig von welchem Thread eine Änderung durchgeführt wurde. Im Gegensatz zur Trace-Äquivalenz wird also von Parallelität und Aktionen in dem Sinne abstrahiert, dass lediglich das beobachtbare Ergebnis am Laufzeitmodell zählt. Das heißt es ist nicht sinnvoll, für jede MAction in einem Prozess eine entsprechende MAction im anderen zu fordern. Dies wäre zu einschränkend, da äquivalente Laufzeitmodelle durch verschiedene (u.U. parallele) Aktionen entstehen. Zum Beispiel lässt sich ein arithmetischer Ausdruck auf verschiedene Weise auswerten, entweder (rekursiv) sequentiell durch parallele Evaluierung seiner Teilausdrücke mit verschiedenen Reihenfolgen etc. Das heißt wir wollen von der Sprachebene des Metamodells (M2) abstrahieren und etwas über das *reine* Modellverhalten (M1) aussagen (s.a. Kapitel 2.5.3).

Daraus folgt, dass wir vielmehr nach einer Entsprechung der „Aktionen im Modell“ suchen, um Bisimulation adäquat auszudrücken. Beispielsweise enthalten Sprachen wie UML *Send*- und *Receive*-Aktionen, die normalerweise durch einen ganzen Satz von MActions spezifiziert sind. Bisimulation sollte deshalb auf der Modellebene (M1) die Frage beantworten, wann eine Abfolge von solchen *Modellaktionen* in zwei UML-Prozessen bisimilar sind. Als generisches Mittel zum Vergleichen bietet sich das Laufzeitmodell und dessen Entwicklung während der Ausführung an, die wir als Grundlage für die Bisimulation heranziehen wollen:

Definition 30 (Bisimulation der Modellausführung) *Zwei Modellausführungen P, Q sind* bisimilar *zueinander (geschrieben: $P \leftrightarrows Q$), wenn Folgendes gilt:*

1. *der initiale Zustand ist modelläquivalent: $\sigma_{0,P} \equiv_M \sigma_{0,Q}$*
2. *für alle induzierten Zustände $\sigma_i(\delta_i)$ der Laufzeitmodelle existiert ein Folgezustand, so dass auch diese äquivalent sind:*
   $\sigma_i(\delta_i)_P \equiv_M \sigma_i(\delta_i)_Q \implies \exists(\sigma_{i+1}(\delta_{i+1})_P \equiv_M \sigma_{i+1}(\delta_{i+1})_Q)$.

Mit der Zustandsskala aus Abschnitt 4.1.2 und 4.1.4 lassen sich daraus mindestens drei verschiedene Bisimulationsäquivalenzen für die Ausführung charakterisieren, abhängig davon wie die Zustände $\sigma_i(\delta_i)$ gebildet wurden:

1. Die Laufzeitmodelle sind nach jedem Mikrozustandsübergang modelläquivalent ($P \leftrightarrows_\mu Q$). Diese strengste Form der Bisimulation erzwingt exakte Übereinstimmung im Ablauf für jede Aktion in der Sprachsemantik und folgt dem kanonischen Modus der MActions.
2. Die Laufzeitmodelle sind nach jedem internen Zustand unter Berücksichtigung der SGTs modelläquivalent ($P \leftrightarrows_{M2} Q$). Enthält eine MActions-Semantik wohldefinierte interne Zustände („Sprachschritte“), so fordern wir Übereinstimmung der Laufzeitmodelle nur für diese Ebene. Beispiele wären hier Transitionsübergänge in SDL/UML oder Anweisungen in Programmiersprachen (vgl. Abschn. 4.1.5), unabhängig von den M1-Modellen.
3. Die Laufzeitmodelle sind nach jedem ausgezeichneten Makrozustand in einem Anwendungsmodell modelläquivalent ($P \leftrightarrows_{M1} Q$). Zum Beispiel könnte die oben erwähnte Abfolge von *Send/Receive*-Aktionen in zwei Prozessen im Anwendungsmodell Makrozustände vor bzw. nach Erhalt von Nachrichten spezifizieren.

Aus der Definition der Zustandshierarchie folgt somit die Bisimulationshierarchie durch Aggregation von Zwischenzuständen (ohne Beweis):

$$P \leftrightarrows_{\mu} Q \implies P \leftrightarrows_{M2} Q \implies P \leftrightarrows_{M1} Q \tag{4.1}$$

Die zweite Äquivalenz $P \leftrightarrows_{M2} Q$ („M2-Bisimulation“) entspricht am ehesten der „klassischen“ Bisimulation (Aktionen der Prozesse müssen äquivalent sein), wohingegen in bisherigen Ansätzen die Unterscheidung M2/M1-Bisimulation außen vor bleibt.[11] Durch die Definition über Makrozustände lässt sich die Granularität der Bisimulation nahezu stufenlos „einstellen“, wenn man zusätzlich angibt, an welcher Stelle ein Makrozustand erreicht wird (s.u.). Bei zwei Programmen, die z.B. einen Sortieralgorithmus für Zahlen über Felder realisieren, könnte man von exakter Verhaltensäquivalenz der Programme (d.h. gleiche Abfolge von Instruktionen, Methodenrufen etc.) über Äquivalenzen der Algorithmen (d.h. Schritte des Sortierens, prinzipieller Ablauf bei der Sortierung) bis hin zur grobgranularen Äquivalenz „liefert gleiches Ergebnis“ (d.h. Zahlen im Feld sind sortiert, unabhängig vom gewählten Algorithmus) wählen. Zu diesem Zweck sind *Zustandsmarker* nötig, die angeben, an welchen Stellen zwei Programme einen äquivalenten Makrozustand erreicht haben. Ein Marker ist eine Relation $\theta\langle \textsc{Object}, \textsc{Object}\rangle$ zwischen zwei Objekten $p \in P$ und $q \in Q$ aus den Modellen P, Q, die auf Instanzebene (M1) kennzeichnet, welche Makrozustände gebildet werden und gemäß der Äquivalenz $P \leftrightarrows_{M1} Q$ verglichen werden sollen. Man kann sich Zustandsmarker ähnlich *Breakpoints* in einer Debuggingumgebung vorstellen, wobei jeweils zwei Breakpoints auf unterschiedliche Programme zeigen und in Relation stehen. Semantisch werden Marker dann bei der Ausführung ausgewertet und erzeugen jeweils im Tracemodell den aggregierten Zustand $\sigma_i(\delta_i)$ aus Def. 30, wenn die Modelle ausgeführt werden.[12]

4.2.4 Bisimulation bei unterschiedlicher operationaler Semantik

Verfolgt man den Gedanken der „M1-Bisimulation“ aus Abschnitt 4.2.4 weiter, lassen sich prinzipiell auch zwei Modellausführungen zu verschiedenen Metamodellen vergleichen. Da wir letztlich nur die Äquivalenz der Laufzeitmodelle zu Grunde gelegt haben, kann Definition 30 erweitert werden, so dass mittels einer Transformation der Laufzeitmodelle (z.B. mit QVT) eine ähnliche Verhaltensäquivalenz definiert werden kann, etwa: $T(\sigma_i(\delta_i)_P) \equiv_M \sigma_i(\delta_i)_Q$, wobei $T : P \to Q$ hier die Transformation der Laufzeitmodelle darstellt. Dies soll hier nicht weiter analysiert werden und lediglich als Ausblick und Anregung für weitere Forschung im Bereich der Verhaltensanalyse und Bisimulation dienen.[13]

4.3 LT-OCL zur Dynamischen Modellanalyse

Während wir im vorigen Kapitel 4.2 den Vergleich zwischen zwei verschiedenen Modellausführungen analysiert haben, sollen im Folgenden die dynamischen Eigenschaften eines Ausführungslaufs betrachtet werden. Dazu definieren wir (exemplarisch)

[11] Man könnte argumentieren, das was hier der Ebene M1 entspricht bereits durch den Abstraktionsgrad der Prozessaktionen einfließt (z.B. Bloom in [93]). D.h. die Prozessdefinitionen würden bereits nur die gewünschten Aktionen beschreiben. Die Bisimulationstheorie behandelt darauf basierend eigentlich nur noch abstrakt die Klasse der durch LTS/GSOS-Regeln erzeugten Prozesse.

[12] Da die Metaklasse eines Objekts möglicherweise mehrere MActivity enthält, muss man eigentlich auf M1-Ebene noch weiter unterscheiden, *wo genau* bei jedem Objekt der Makrozustand erreicht wird, d.h. θ ist eigentlich eine Relation zwischen Tupel (Object,MActivity).

[13] Ein Beispiel für solch einen Vergleich mit *QVT Relations* ist in [119] beschrieben (das Papier wurde bei der SLE 2008 abgelehnt und stattdessen als Whitepaper ohne Review veröffentlicht)

Syntax und Semantik der *Linear Time Temporal Object Constraint Language* (LT-OCL [120]), die zur Spezifikation von dynamischen Eigenschaften dient. Temporallogiken ermöglichen generell eine formale Ausdrucksweise über Eigenschaften, die sich zeitlich während eines Simulationslaufs ergeben. Wie Kapitel 2.7 zeigt können dabei mittels der prädikatenlogische Variablenbindung komplexe Zusammenhänge recht kompakt formuliert werden. Ob prinzipiell eine Temporallogik für dynamische Analysen geeignet ist oder andere Formen von Algorithmen zur Auswertung von Laufzeitdaten besser anwendbar sind, soll hier nicht beantwortet werden. Stattdessen dienten die folgenden Definitionen als Studienobjekt dem Zweck, Erkenntnisse zum metamodellunabhängigen Ansatz einer dynamischen Modellanalyse zu gewinnen und dessen Grenzen auszuloten.

Konzeptionell führen wir dazu drei unäre temporale Operatoren `next`, `always`, `eventually` sowie den binären Temporaloperator `until` als Erweiterungen der OCL ein. Diese entsprechen den in Abschnitt 2.7.1 beschriebenen klassischen Operatoren $\bigcirc$, $\Box$, $\Diamond$ und $\mathcal{U}$. Da LT-OCL-Formeln im Kontext eines Metamodells mit operationaler Semantik interpretiert werden sollen, binden wir die Auswertungssemantik an das generische Trace-Metamodell und dessen implizierte Zustände (vgl. Abschnitt 4.1.3). Dadurch ist die Semantik von LT-OCL-Ausdrücken abhängig von einem konkreten Metamodell, der Aktionssemantik dieses Metamodells und den darin enthaltenen Zustandsdefinitionen. Die Anwendung und Probleme diskutieren wir anschließend entlang einer Fallstudie in Kapitel 4.4.1.

4.3.1 Übersicht der LT-OCL

Modellverifikation mit LT-OCL ist eng verknüpft mit dem Laufzeitmodell und dessen Änderungen während der Simulation, weil alle dynamischen Informationen im Laufzeitmodell hinterlegt sind (vgl. Abschnitt 4.1.1). Dies führt dazu, dass temporale Ausdrücke fast ausschließlich über Objekte der Klassen des Laufzeitmodells formuliert sind. Durch die Untermenge Essentiel OCL, welche als Basis der LT-OCL dient, sind OCL-Annotationen prinzipiell an Klassen, Pakete und Operationen möglich. Dies resultiert aus dem *package merge* des MOF-Metamodells, bei dem der sogenannte *contextual classifier* (über das Schlüsselwort `context` gekennzeichnet) nicht weiter eingeschränkt wird. Für LT-OCL schränken wir hingegen diesen Kontext auf Klassen- und Paketinvarianten ein, da neben dem Blickwinkel auf die Laufzeitgrößen weiterhin ein Ausdruck immer über einen gesamten Trace ausgewertet wird. Für gerufene Operationen existiert keine „partielle Semantik“. [14]

Die Mächtigkeit einer Temporallogik zeigt sich anhand von einigen Beispielen, die die drei typischen Arten von temporalen Aussagen klassifizieren: *safety*, *liveness* und *fairness*. Diese Liste stellt keinen Anspruch auf Vollständigkeit, zeigt aber gängige „Muster“ von Kombinationen temporaler Operatoren zu Ausdrücken und bietet so gleichzeitig eine informelle Einführung in die Logik LT-OCL. Die einfachste Form dynamischer Eigenschaften stellen *safety properties* dar, also Invarianten, die während einer gesamten Ausführung erfüllt sein sollen. Ein Beispiel für eine Programmiersprache mit dynamischer Speicherallokation wäre:

always(heap.size $<$ limit)

Dazu müsste `heap` z.B. durch einen *let*-Ausdruck an ein entsprechendes Objekt gebunden werden, dass den Speicher repräsentiert und die Variable `limit` einen Wert

[14] Für UML definiert OCL bereits mit @*pre* und @*post* ein Zustandsbezug für Vor- und Nachbedingungen, der einen Zugriff auf ausgewählte „Zeitpunkte“ ermöglicht

erhalten, der die obere Schranke spezifiziert. Bei der Auswertung über einen Simulationslauf wird die Teilformel innerhalb des **always**-Operator in jedem Zustand überprüft.

Eine zweite häufig anzutreffende Kombination ist die eines *liveness properties*, bei der eine always/eventually-Kombination genutzt wird (vgl. 2.7):

```
always(event.send implies eventually(event.received))
```

Bei der Auswertung wird bei jedem wahr werden des Teilausdrucks `event.send` eine Prüfung zu jedem weiteren Zustand durch **eventually** initiiert. Dies geht so lange, bis entweder `event.received` wahr wird oder aber die Ausführung ans Ende gelangt und den gesamten Ausdruck als falsch auswertet. Verallgemeinert kann dadurch eine Änderung an Werten verifiziert werden, wobei der Zeitpunkt des Eintretens „irgendwann“ erfolgen muss. Weiterhin ist die Bindung der (hier freien) Variable `event` entscheidend. So kann z.B. ein OCL **forAll**-Operator in Zusammenhang mit der Objektmenge **Event.allInstances()** benutzt werden, um den Ausdruck über alle jemals gesendeten Events zu garantieren.

Für parallele Ausführungen und Nebenläufigkeit von Prozessen stehen besonders Fairness-Eigenschaften im Analysefokus. Anhand des klassischen *Dining Philosophers*-Beispiels[15] lassen sich diese demonstrieren. Eine *strong fairness*-Eigenschaft ist, wenn für eine beliebig oft eintretende Bedingung immer ein Folgeereignis eintritt. Zum Beispiel soll ein Philosoph, der beliebig oft zwei Stäbchen erhält auch in der Folge in den Zustand „Essen“ gelangen:

```
always(eventually(p.sticks->size() = 2))
    implies always(eventually(p.state = #Eating))
```

Die Variable p steht hier stellvertretend für einen der Philosophen, sein aktueller Zustand sei im Attribut `state` als Enumeration mit den Werten **{Thinking, Eating}** und dem Mengenattribut `sticks` für die momentan gegriffenen Stäbchen gegeben. Dagegen kann eine *weak fairness*-Bedingung für das gleiche Beispiel beschrieben werden, die das Zurücklegen von beiden Stäbchen ausdrückt:

```
eventually(always(p.state = #Thinking))
    implies always(eventually(p.sticks->size() = 0))
```

4.3.2 OCL Syntaxerweiterungen für temporale Operatoren

Im OCL Standard ist die Konkrete Syntax durch eine attributierte EBNF Grammatik inklusive Abbildung auf das OCL Metamodell (als abstrakte Syntax) definiert (vgl. Abschnitt 9 in [17]). Deshalb bedürfen alle LT-OCL-Konzepte einer Erweiterung sowohl der Grammatikregeln, des Metamodells als auch der Abbildungsregeln in dieses erweiterte OCL-Metamodell.

Abbildung 4.2 zeigt das LT-OCL-Metamodell mit den Erweiterungsklassen für temporale Operatoren (neue Klassen grau unterlegt). Die der EBNF Grammatik weitestgehend folgende, syntaxorientierte Vererbungshierarchie des OCL Metamodells erlaubt eine einfache Erweiterung der Sprachkonzepte. Die Klassen im einzelnen:

TemporalExp Stellt die abstrakte Basisklasse aller temporalen Ausdrücke dar. Über die Assoziation `expression` wird immer ein boolescher Ausdruck erwartet (s.a. Produktionsregel in der Grammatik). Alle Subklassen von `TemporalExp` liefern einen boolschen Wert.

[15] vgl. [121] für ein Metamodell mit operationaler Semantik

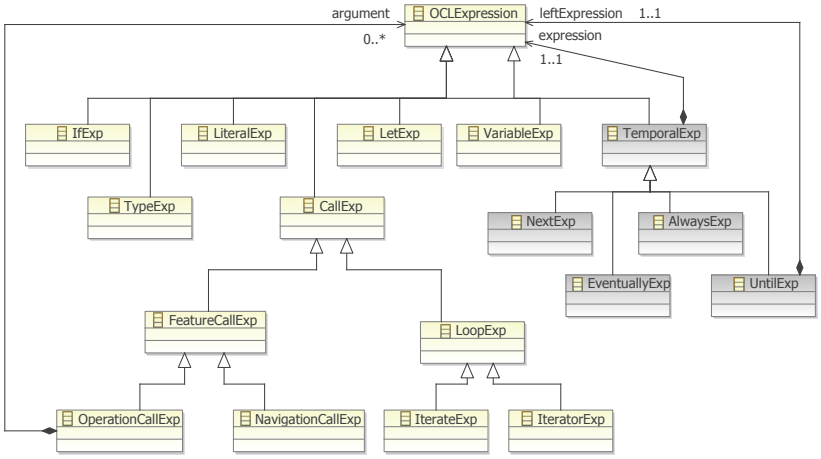


Abbildung 4.2: LT-OCL-Metamodell mit Erweiterungen für Temporale Operatoren

NextExp Entspricht dem `next` Operator und gibt an, dass `expression` im nächsten Zustand wahr sein muss, damit der gesamte Ausdruck wahr wird.

AlwaysExp Repräsentiert den `always` Operator und verlangt von `expression`, das diese ab dem Moment der Evaluierung für alle folgenden Zustände wahr sein muss.

EventuallyExp Metaklasse die `eventually` repräsentiert. Um wahr zu werden, muss `expression` in mindestens einem Folgezustand wahr werden.

UntilExp Entspricht dem binären `until` Operator und behandelt neben `expression` die über `leftExpression` angegebene Vorbedingung. Damit der Gesamtausdruck wahr wird, muss `leftExpression` in jedem Zustand wahr sein bis `expression` wahr wird.

Die EBNF Grammatik des OCL-Standards benutzt Attribute an den Produktionsregeln zur Beschreibung der Abbildung ins OCL-Metamodell. Dabei erhält eine Regel immer das synthetisierte Attribut `ast` (Abk. für *abstract syntax tree*), das (1) die beim Parsen zu instanziierende Metaklasse angibt (also als Typ des AST Knotens) und (2) über eine Punkt-Notation Zugriff und Navigation im Metamodell erlaubt. Der Wert der synthetisierten Attribute auf der linken Seite einer Produktionsregel leitet sich somit immer aus Attributen auf der rechten Seite einer Regel ab. Zusätzlich besteht über ein ererbtes Attribut `env` (Abk. für *environment*) Zugriff auf den aktuell gültigen Namensraum und der aktuellen Belegung von Bezeichnern. Der Typ des Attributs ist eine im OCL-Standard zum Parsen definierte Klasse `Environment`, welche Methoden zum Nachschauen von bereits geparsten Werten erlaubt (vgl. Abschnitt 9.1 in [17]).

Um nicht alle Definitionen des Standards zu wiederholen, geben wir im Weiteren nur die neuen Regeln an und verweisen den Leser auf [17]. Sollten vorhandene Regeln verändert oder eingeschränkt werden, werden diese explizit aufgeführt. Auf oberster Ebene der Grammatikregeln fügen wir eine neue Regel `[G]` für temporale Operatoren hinzu:

```
[A] OclExpressionCS ::= PropertyCallExpCS
[B] OclExpressionCS ::= VariableExpCS
[C] OclExpressionCS ::= LiteralExpCS
[D] OclExpressionCS ::= LetExpCS
-- [E] OclExpressionCS ::= OclMessageExpCS
```

```
-- nicht in Essential OCL
[F] OclExpressionCS ::= IfExpCS
[G] OclExpressionCS ::= NextExpCS | AlwaysExpCS |
                        EventuallyExpCS | UntilExpCS
```

Die vier neuen Nichtterminal-Symbole definieren sich wie folgt:

```
[A] NextExpCS ::= 'next' '(' OclExpressionCS ')'
[B] AlwaysExpCS ::= 'always' '(' OclExpressionCS ')'
[C] EventuallyExpCS ::= 'eventually' '(' OclExpressionCS ')'
[D] UntilExpCS ::= '(' OclExpressionCS[1] ')' 'until'
                            '(' OclExpressionCS[2] ')'
```

Neben Token in einfachen Anführungszeichen wird die Indizierung in `[D]` für eine Unterscheidung der geparsten Werte zur weiteren Zuordnung zu dem synthetisierten Attribut `ast` verwendet. Dieses wird bei der Definition der AST-Struktur wie folgt verwendet:

```
NextExpCS.ast : NextExp
[A] NextExpCS.ast.expression = OclExpressionCS.ast
AlwaysExpCS.ast : AlwaysExp
[B] AlwaysExpCS.ast.expression = OclExpressionCS.ast
EventuallyExpCS.ast : EventuallyExp
[C] EventuallyExpCS.ast.expression = OclExpressionCS.ast
UntilExpCS.ast : UntilExp
[D] UntilExpCS.ast.leftExpression = OclExpressionCS[1].ast
[D] UntilExpCS.ast.expression = OclExpression[2].ast
```

Durch das Attribut `ast`, getypt auf Klassen des Metamodells, wird die Brücke von konkreter zu abstrakter Syntax geschlagen. Wie in den Regeln `[A]` bis `[D]` ersichtlich, instanziiert `ast` die Klassen `NextExp`, `UntilExp` etc. und die Referenz `expression` wird konsistent jeweils auf die Werte der Kindausdrücke vom Typ `OclExpressionCS` gesetzt. Als Besonderheit des `until`-Ausdrucks wird die linke Seite der geparsten Werte (durch `OclExpressionCS[1].ast`) repräsentiert) der `left-Expression` zugeordnet, die rechte Seite der `expression`-Referenz.

Zusätzlich zu diesen Konstruktionsregeln schränken wir von allen vier Temporalausdrücken den Typ der umklammerten Kindausdrücke auf boolesche Ausdrücke ein:

```
NextExpCS.ast.expression.type.name = 'Boolean'
AlwaysExpCS.ast.expression.type.name = 'Boolean'
EventuallyExpCS.ast.expression.type.name = 'Boolean'
UntilExpCS.ast.leftExpression.type.name = 'Boolean'
```

4.3.3 LT-OCL-Semantik

Die Semantik von LT-OCL-Ausdrücken ist abhängig von einem konkreten Metamodell inklusive Aktionssemantik. Da sich während der Ausführung Modellzustände als Traces aufzeichnen lassen, lässt sich mittels der Tracedefinition aus Abschnitt 4.1.3 und den induzierten Zuständen aus Abschnitt 4.1.4 eine generische Semantik angeben, die lediglich indirekt an die MActions Aktionssemantik gekoppelt ist. Daraus ergeben sich zwei Vorteile:

1. Die Sprachsemantik von LT-OCL lässt sich in klassischer Weise über Pfade von Zuständen angeben. Dies erleichtert sowohl die Definition als auch das Verständnis der temporalen Ausdrücke.
2. LT-OCL ist durch das Tracemodell von den MActions entkoppelt und lässt sich bei anderen dynamischen Semantiken für Metamodelle verwenden, vorausgesetzt deren Ausführungssemantik kann durch Zustände beschrieben werden.

Bisher haben wir Modellzustände in Definition 27 als induzierte Zustände des Laufzeitmodells angegeben. Um die LT-OCL-Semantik formal zu fassen, soll nun eine Abbildung auf den algebraischen Kalkül aus Abschnitt 2.4.2 und 2.6.1 folgen, indem wir induzierte Zustände auf das Zustandsmodell $\sigma(\mathcal{M})$ abbilden.

Definition 31 (Zustandssequenz) *Sei $\mathcal{M}$ ein Objektmodell mit*

$$\mathcal{M} = (C_{\text{LASS}}, A_{\text{TT}_c}, O_{\text{P}_c}, A_{\text{SSOC}}, associates, roles, multiplicities, \prec) \tag{4.2}$$

gemäß Definition 7 und $\sigma(\mathcal{M})$ der initiale Zustand des Laufzeitmodells mit

$$\sigma_0(\mathcal{M}) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC}) \tag{4.3}$$

Dann bezeichnen wir mit $\sigma_i(\gamma_i)$ den (induzierten) Systemzustand zum Zeitpunkt i, der sich durch Abbildung von γ_i gemäß Definition 8 ergibt. Dazu bezeichnen wir mit $\pi = \sigma_0, \ldots, \sigma_t$ $(t \in \mathbb{N})$ die Zustandssequenz *einer Modellausführung bis zum Zeitpunkt t. Wir nennen π* vollständig *in Bezug auf eine Modellausführung, wenn die komplette geordnete Menge $\gamma_0, \ldots, \gamma_n (n \in \mathbb{N})$ aller* `Traces` *in π enthalten ist (d.h. wenn $t = n$).*

Zusammengefasst lässt sich feststellen, dass durch diese Definition aus jedem Zustand des Laufzeitmodells über Umweg des induzierten Zustands γ_i im `Trace` der formalisierte Systemzustand $\sigma_i(\gamma_i)$ wird. Der Trace ist also lediglich Mediator, um der Aktionssemantik a priori keinen Zustand „aufzuzwingen“. Darüber hinaus sei angemerkt, dass wir mit der Terminologie der Historie der formalen (Transitions-)Systeme Rechnung tragen und den algebraisch formalen Teil mit *Systemzustand* bezeichnen, wohingegen beim Modell schlicht von Zustand die Rede ist.

Als nächstes muss die Auswertung von OCL-Ausdrücken in Bezug auf die Evaluierungsumgebung erweitert werden. In Abschnitt 2.6.1 wurde bereits die Interpretationsfunktion $I[\![expr]\!] : Env \to I(t)$ für eine statische Umgebung Env eingeführt. Diese Umgebung gilt es im Hinblick auf Zustandssequenzen zu erweitern.

Definition 32 (Erweiterte Umgebung) *Sei $Expr_t$ die Menge aller OCL-Ausdrücke von Typ t mit den temporalen Erweiterungen aus Abschnitt 4.3.2 und Var_t die Menge aller Variablen. Eine* erweiterte Umgebung *$\varepsilon = (\pi, \beta, i)$ besteht aus einer Zustandssequenz π und einer Projektion $\beta : Var_t \times \mathbb{N} \to I(t)$, die Variablen ihre Werte zum Zeitpunkt i zuordnet.*

Für eine bessere Leserlichkeit schreiben wir im Folgenden ε_i anstelle von $\varepsilon = (\pi, \beta, i)$, wenn sich π und β aus dem Kontext erschließen.

Definition 33 (Semantik von LT-OCL-Ausdrücken) *Sei $\pi = \sigma_0, \ldots, \sigma_n$ $(n \in \mathbb{N})$ eine Sequenz von Systemzuständen, $e \in Expr_t$ ein LT-OCL-Ausdruck und Env die erweiterte Umgebung $\varepsilon = (\pi, \beta, i)$. Dann ist die Auswertung des LT-OCL-Ausdrucks e durch die Interpretationsfunktion $I[\![e]\!] : Env \to I(t)$ wie folgt definiert:*

i. *Standard OCL-Ausdrücke werden wie im Annex zur formalen Semantik in [17], Appendix A, Definition A.14 - A.30 ausgewertet,*[16] *unter Berücksichtigung der erweiterten Umgebung* $I[\![e]\!](\varepsilon)$*, welche den Auswertungszeitpunkt* i *enthält. Dies bedeutet im Einzelnen:*
   (1) *Variablenzuweisungen erhalten den Wert der Umgebung zum Zeitpunkt* i*:* $I[\![v]\!](\varepsilon) = \beta(v, i)$
   (2) *Iteratorausdrücke iterieren immer über konsistente Mengen die zum Zeitpunkt* i *ermittelt worden sind*
   (3) *Operationsrufe werden als seiteneffektfrei angesehen und verändern nicht das Laufzeitmodell (d.h. sie interferieren nicht). Dadurch ist eine Auswertung einer Operation konsistent zum Zeitpunkt* i

ii. *Ein 'next' Ausdruck wird verzögert im nächsten Systemzustand der Sequenz* π *ausgewertet:*

$$I[\![next(e)]\!](\varepsilon) = \begin{cases} true & \text{wenn } I[\![e]\!](\varepsilon_{i+1}) = true \\ false & \text{wenn } I[\![e]\!](\varepsilon_{i+1}) = false \\ \bot & \text{sonst} \end{cases}$$

iii. *Ein 'until' erzwingt, dass ein Ausdruck* e_1 *in allen Systemzuständen wahr ist, bis der zweite Ausdruck* e_2 *wahr wird:*

$$I[\![(e_1)\ until\ (e_2)]\!](\varepsilon) = \begin{cases} true & \text{gdw. } \exists k.i < k, \text{ so dass } \forall i \leq u < k : I[\![e_1]\!](\varepsilon_u) = true \\ & \text{und } \forall j \geq k : I[\![e_2]\!](\varepsilon_j) = true \\ false & \text{gdw. für ein } i \leq u < k\ I[\![e_1]\!](\varepsilon_u) = false \text{ oder} \\ & I[\![e_2]\!](\varepsilon_j) = false \\ \bot & \text{sonst} \end{cases}$$

iv. *Ein 'always' Ausdruck ist genau dann wahr, wenn er in allen Systemzuständen wahr ist (ab Auswertungszeitpunkt):*

$$I[\![always(e)]\!](\varepsilon) = \begin{cases} true & \text{wenn } \forall i \in 0, ..., n.I[\![e]\!](\varepsilon_i) = true \\ false & \text{wenn } \exists i \in 0, ..., n.I[\![e]\!](\varepsilon_i) = false \\ \bot & \text{sonst} \end{cases}$$

v. *Ein 'eventually' Ausdruck ist genau dann wahr, wenn ein Systemzustand in* π *folgt, in dem* e *wahr wird:*

$$I[\![eventually(e)]\!](\varepsilon) = \begin{cases} true & \text{if } \exists i \in 0, ..., n.I[\![e]\!](\varepsilon_i) = true \\ false & \text{if } \forall i \in 0, ..., n.I[\![e]\!](\varepsilon_i) = false \\ \bot & \text{otherwise} \end{cases}$$

Ein LT-OCL-Ausdruck $e \in Expr_t$ ***ist wahr für*** π *gdw.* $I[\![e]\!](\pi, \beta, 0) = true$*.*

Diese Definition sollte auf den ersten Blick nicht überraschen, lässt sich doch eine große Ähnlichkeit zu der algebraischen LTL-Semantik aus Abschnitt 2.7 feststellen. Die dort versprochene Diskussion zu den Besonderheiten der Auswertung soll nun folgen.

Diskussion zur LT-OCL-Semantik

Zunächst ist ein LT-OCL-Ausdruck immer über eine endliche Sequenz π von Systemzuständen definiert. Theoretisch ließe sich die Definition auch auf unendliche

[16] wir geben an dieser Stelle nur vom OCL Standard abweichende/erweiterte Definitionen an

Pfade ausweiten, jedoch hat das für die praktische Anwendung im „normalen“ (d.h. endlichen) Simulationsexperiment keine Bedeutung; schließlich möchte man am Ende einer terminierten Modellausführung wissen, ob ein Ausdruck für diesen Ablauf wahr ist oder nicht. Darüber hinaus ergibt sich durch die Erweiterung auf unendliche Pfade die Problematik der Entscheidbarkeit von prädikatenlogischen Temporallogiken (vgl. [122]).

Betrachtet man die per Definition eingeschränkte Gültigkeit eines LT-OCL-Ausdrucks auf einen Pfad, so lässt sich durchaus eine Ausweitung auf *alle* Ausführungsläufe in Erwägung ziehen. Man könnte somit die Gültigkeit einer Formel erweitern, z.B indem man sie in Bezug auf das Modell*verhalten* definiert. Das heißt ein LT-OCL-Ausdruck wäre gültig für ein Modellverhalten gdw. er auf allen möglichen Pfaden wahr ist. Dies entspricht einer implizierten Kripke-Struktur, welche aus den erreichbaren Systemzuständen des Modells besteht. Bei genauerer Betrachtung ist diese Erweiterung jedoch nicht ohne Weiteres machbar, da z.B. der Zustandsraum des Laufzeitmodells in der Regel unendlich ist. Im Hinblick auf ein „*Modellchecking*“ (im Sinne einer formalen Verifikation über dem vollständigen Zustandsraum) böte sich diese Definition dennoch als Basis an, um durch gezielte Suche im Zustandsraum zu ermitteln, ob ein Ausdruck wahr oder falsch ist.

Eine Besonderheit der LT-OCL besteht in der Variablenbindung und der erweiterten Umgebung, die zusammengenommen einer detaillierten Diskussion bedürfen. Betrachten wir folgende Invariante einer fiktiven Metaklasse `State` mit einem Attribut `active` vom Typ Boolean:

context State **inv**:
eventually(**always**(self.active **and** var=42))

Diese Temporale Formel besteht neben einer typischen `eventually/always` Kombination aus der eigentlichen Konjunktion von `self.active` und der freien Variable `var`, die nicht durch einen Iterator oder `let` gebunden ist. Durch Anwendung der Definition 33 ergibt sich der Wert von `var` zu einem Zeitpunkt t aus: $I[\![var]\!](\varepsilon) = \beta(var, t)$, d.h. möglicherweise liefert die Umgebung ε zu verschiedenen Zeitpunkten t unterschiedliche Werte. Die Temporale Formel fordert also, dass sobald der Zustand einen Wert von `self.active=true` hat und `var` den Wert `42`, diese sich nicht mehr ändern.

Ein Beispiel für eine gebundene Variable zeigt die LT-OCL-Formel, die eine (nicht genauer spezifizierte) Metaklasse `Event` adressiert, diese hat zwei boolesche Attribute `send` und `receive`:

always (Event.allInstances()−>forAll(e | e.send **implies eventually** (e.receive)))

Auf den ersten Blick könnte `e` wie eine freie Variable wirken, allerdings stellt sie tatsächlich den OCL `forAll`-Iterator dar. Durch diese Form wird ein expliziter Zugriff auf iterierte Werte/Objekte ermöglicht. Im Hinblick auf die zeitliche Auswertung von `e` sollte deutlich werden, dass sich die Teilausdrücke `e.send` und `e.receive` i.Allg. auf unterschiedliche Zeitpunkte beziehen. Das heißt, sobald für ein Event e_1 `send` wahr wird, muss für *dasselbe* Event e_1 irgendwann später `receive` wahr werden. Dabei wird jedoch die Menge aller `Event` Instanzen durch `allInstances` zu jedem Zeitpunkt erneut ausgewertet. Anders würde es sich verhalten, wenn wir den `always` Ausdruck in die `forAll`-Iteration hineinziehen:

Event.allInstances()−>forAll(e | **always**(e.send **implies eventually** (e.receive)))

Bei dieser Auswertung würden nur diejenigen Events berücksichtigt, die zum Zeitpunkt t_0 existieren (d.h. genau alle Instanzen, die im initialen Laufzeitmodell vorhanden sind).

4.4 Fallstudien

Für diese Arbeit wollen wir uns auf zwei Fallstudien beschränken, die die wesentlichen Aspekte des Zusammenspiels von MActions und LT-OCL verdeutlichen. Neben den im Folgenden Abschnitt 4.4.1 und im Anhang A gezeigten Sprachen (publiziert in [120],[20]) wurden die Konzepte im Kontext von domänenspezifischen Sprachen z.B. anhand von Beispielsprachen in [123] und [119] untersucht.

4.4.1 Timed Automata

Dieser Abschnitt beschreibt eine beispielhafte Anwendung von LT-OCL auf ein Metamodell mit MActions, dem Dampfkessel-Kontrollsystem (engl. *steam boiler control system*). Dieses Beispiel wurde Mitte der neunziger Jahre im Rahmen eines Dagstuhl Seminars als ein komplexes Referenzproblem vorgestellt, um verschiedene formale Spezifikationstechniken zu vergleichen. Die Problembeschreibung lautet zusammengefasst wie folgt (Details siehe [124]):

> Es soll ein Kontrollprogramm für einen Regelkreis spezifiziert werden, das einen Dampfkessel steuert. Der Dampfkessel besteht aus einem Wassertank in dem Wasser erhitzt wird, das als Wasserdampf über ein Ventil eine Turbine antreibt. Der Wasserzufluss ist durch vier Pumpen und ein zusätzliches Ventil geregelt, die alle vom Kontrollprogramm angesteuert werden können, um den Wasserstand im Tank zwischen einem Minimal- und Maximalpegel zu halten. Der Wasserstand kann dabei durch einen Messsensor vom Programm ausgewertet werden.
>
> Sollte es zu einem kritischen Wasserstand ober- oder unterhalb der Pegelgrenzen kommen, muss binnen 5 Sekunden durch Regelung der Pumpen oder des Ventils ein normales Niveau wiederhergestellt werden, ansonsten droht der Dampfkessel oder die nachgeschaltete Turbine Schaden zu nehmen. Sollte dies aus irgendwelchen Gründen nicht möglich sein (z.B. keine ausreichende Wasserzufuhr, Fehlfunktion des Ventils), muss das System in einen Nothalt versetzt werden.

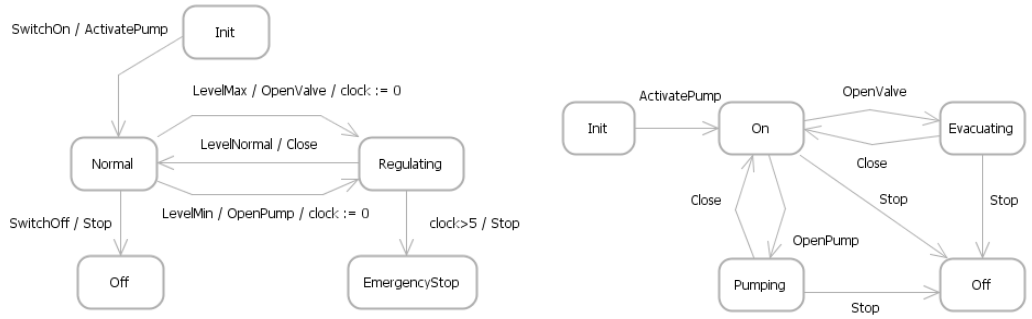
Im Folgenden werden wir die Sprache der *Timed Automata* (TA) beschreiben, mit der die Spezifikation adäquat repräsentiert werden kann. Die Sprache ist eine vereinfachte Form der von Alur, Dill et al. entwickelten und im Werkzeug Uppaal implementierten Modellierungssprache für automatische Verifikation von Echtzeitsystemen (vgl. [125][126]). Anschließend folgt dann eine Analyse der Funktionsweise mittels LT-OCL.

Steam Boiler als Zustandsautomat

Für die Fallstudie werden wir das Originalproblem zunächst auf die wesentlichen Bestandteile reduzieren. Dazu begrenzen wir die Anzahl der Pumpen auf eine und abstrahieren von Details der Wasserstands- und Dampfstrahlmessung sowie der Kommunikation.[17] Zur weiteren Vereinfachung verbinden wir die Pumpe und das Ventil zu einer *Tankeinheit*, die sowohl Pumpe als auch Ventil ansteuert. Davon separat spezifizieren wir das eigentliche *Kontrollprogramm*, dass die Tankeinheit über-

[17] In der Originalbeschreibung finden sich darüber hinaus weitergehende Details zu Betriebsparametern, Wartungszuständen, etc. Das Ventil dient eigentlich nur der initialen Evakuierung des Wassertanks

wacht. Abbildung 4.3 zeigt zwei Zustandsautomaten, die die beiden Komponenten beschreibt.



(a) Kontrollprogramm (b) Tankeinheit

Abbildung 4.3: Das Steam Boiler-Kontrollsystem als Zustandsautomaten: 4.3(a) Kontrollprogramm, 4.3(b) Pumpensteuerung

Die Zustandsautomaten sind über das Metamodell in Abbildung 4.4 definiert. Jeder Zustandsautomat ist repräsentiert durch eine Instanz der Klasse `StateMachine`, die ihre Zustände über `states` enthält. Diese umfassen wiederum alle ausgehenden Transitionen via `outgoingTransitions`. Das Laufzeitmodell besteht aus den grau markierten Klassen `Event`, `EventBroker` sowie `StateMachineInstance` (im folgenden mit `SMI` abgekürzt). Letztere repräsentiert als Instanz der Klasse `StateMachine` den 'Zustand' des Automaten zur Laufzeit. Hier findet die *instanceOf*-Beziehung Verwendung, welche uns mittels der implizierten `metaObject`-Referenz zur Laufzeit Zugriff auf Instanzen der Klasse `StateMachine` erlaubt. Als weitere Zustandsgrößen enthält diese Klasse die Referenz auf den aktuellen Zustand in `activeState` und die interne Uhr des Automaten mittels `clock`. `SMI` Instanzen werden an einen `EventBroker` über die `clients`/`broker`-Assoziation gebunden und können dadurch `Events` erhalten.

Der Verständlichkeit halber nutzen wir die Standard UML-Notation als Syntax für Zustände und Transitionen und setzen ein intuitives Verständnis für die Abbildung zwischen konkreter und abstrakter Syntax voraus. Als Besonderheit verwenden wir für Beschriftungen an Transitionen die folgende Tripelnotation:

```
<guard> / <action> / <clock-reset>
```

wobei `<action>` und `<clock-reset>` optional sind. Die Werte von `<guard>` und `<action>` werden direkt auf entsprechende Attribute der Klassen `Transition` abgebildet unter Berücksichtigung der folgenden Regeln:

(1) `<clock-reset>` kann nur den Wert '`clock := 0`' annehmen, welcher direkt zu `resetClock=true` wird (anderseits ist `resetClock` immer `false`)
(2) alle Werte von `<guard>` die mit `'clock'` beginnen bezeichnen den Wert des Attributs `timeLimit`

Diese Einschränkungen werden lediglich aus Gründen der Einfachheit eingeführt, um die MActions Semantik dem Beispiel entsprechend möglichst minimal zu halten.[18]

Konzeptionell erhält jeder Zustandsautomat eine lokale Uhr. Alle Uhren werden während der Ausführung kontinuierlich in diskreten Schritten erhöht. Eine Transition kann die Uhr des eigenen Automaten durch `clock := 0` zurücksetzen oder auf

[18] Andernfalls hätte eine detaillierte Spezifikation der Syntax einer *expression language* angegeben werden müssen, die nichts wesentliches zur Fallstudie beigetragen hätte

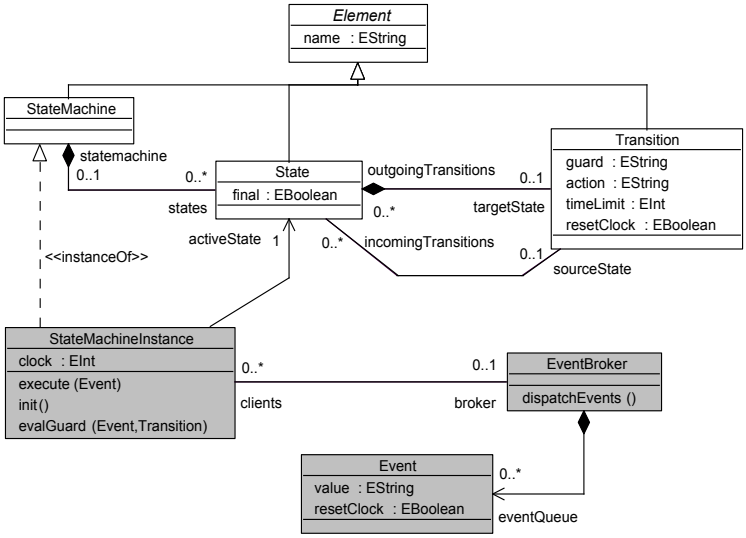


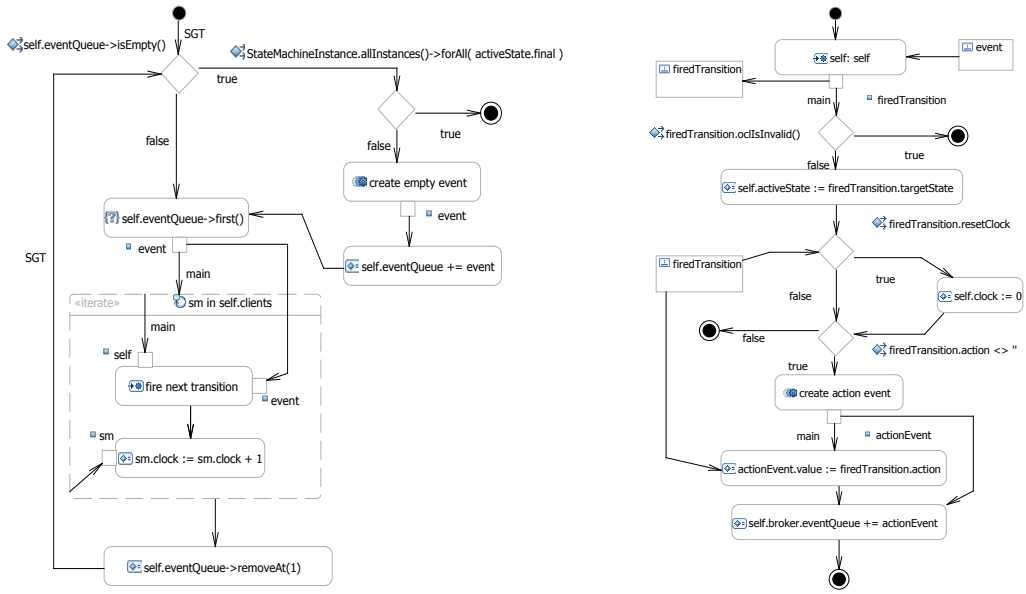
Abbildung 4.4: Simple StateMachine Metamodel

Änderungen der Uhr reagieren. Das heißt eine Transition wird geschaltet, sobald ein (absolutes) Zeitlimit (als `guard` angegeben) erreicht wird. Demnach lässt sich das Verhalten der beiden Zustandsautomaten intuitiv wie folgt beschreiben:

- Der Zustandsautomat des Kontrollprogramms aus Abbildung 4.3 schaltet nach erhalten des `SwitchOn`-Ereignisses in den operativen Zustand `Normal` und emittiert dabei das `ActivatePump`-Ereignis. Dieses führt bei der Tankeinheit zum Schalten einer Transition in den Folgezustand `On`.
- Sobald ein `LevelMin`- oder `LevelMax`-Ereignis erhalten wird, welches ein Verlassen des Wasserstandes aus dem Regelbereich beschreibt, schaltet das Kontrollprogramm in den aktiv regulierenden Zustand `Regulating` und beginnt mit der Regulierung indem entweder ein `OpenValve`- oder `OpenPump`-Ereignis gesendet wird. Zur gleichen Zeit wird die lokale Uhr zurückgesetzt, um die Zeit zu messen, die reguliert wird.
- Wird ein `LevelNormal`-Ereignis erhalten, dass das Erreichen eines normalen Wasserpegels anzeigt, schaltet das Kontrollprogramm zurück zu `Normal` und sendet dabei ein `Close` an die Pumpe bzw. das Ventil. Sollte allerdings innerhalb von 5 Sekunden kein `LevelNormal`-Ereignis empfangen worden sein, so hält das ganze System im Endzustand `EmergencyStop`.

Diese Systembeschreibung ist soweit gesehen unvollständig, da keine der beiden Komponenten jemals die Ereignisse `LevelNormal`, `LevelMin` und `LevelMax` emittiert. Wir werden in einem zweiten Schritt über die Details der Ereignisse sprechen, zunächst muss die operationale Semantik der TA definiert werden.

Die MActions werden durch die in 4.4 dargestellten `MOperations` ins Laufzeitmodell integriert. Die Hauptausführungsschleife stellt die in `dispatchEvents` gezeigte bedingte Schleife dar (vgl. 4.5(a)). Unter der Annahme, dass alle `SMI` als Klienten mittels `clients` zu einer `EventBroker`-Instanz verbunden sind, werden diese reihum mit Ereignissen adressiert und erhalten die Möglichkeit, auf diese zu reagieren. Durch dieses Broadcast-Verfahren erhalten alle Zustandsautomaten dasselbe Ereignis und können nicht nur Schalten, sondern im Gegenzug neue Ereignisse emittieren und an das Ende der `eventQueue` anhängen (FIFO). Dies ist wie folgt realisiert (vgl. Abb. 4.5(a)):



(a) EventBroker#dispatchEvents (b) StateMachineInstance#execute

Abbildung 4.5: Ereignisverarbeitung und Schaltsemantik der Timed Automata

Ein Entscheidungsknoten überprüft zunächst, ob die `eventQueue` leer ist. Sollte dies der Fall sein, prüft eine weitere Bedingung, ob alle Zustandsautomaten ihren Endzustand erreicht haben.[19] Solange noch nicht alle Automaten diesen erreicht haben, wird ein Defaultereignis erzeugt (s. MCREATE-Aktion `create empty event`) und in die `eventQueue` eingereiht (s. mehrwertige Zuweisungsaktion mit '`+=`'-Operator), um den Kanal stets mit mindestens einem Ereignis gefüllt zu halten. Die nächste Aktion holt das erste Element dieser Liste und reicht es an eine Iterationsaktion, die über alle `clients` iteriert. Während der Iteration werden immer zwei Aktionen ausgeführt:

fire next transition ist eine Invokationsaktion mit dem Ziel `execute(event : Event)`, bei der die Iterationsvariable `sm` als `self` des Aufrufs verwendet wird und `event` als Parameter übergeben wird. Die `execute`-Operation ist in Abb. 4.5(b) gezeigt und beschreibt einen einzelnen Schaltschritt eines Zustandsautomaten.

sm.clock := sm.clock + 1 inkrementiert das `clock`-Attribut des aktuellen Zustandsautomaten `sm`.

Nach traversieren aller Zustandsautomaten wird das Ereignis als abgearbeitet angesehen und von der Ereignisliste gelöscht (`removeAt`-Aktion). Zusammengefasst modelliert dieser Satz an Aktionen bereits die Kernsemantik eines verlustlosen Ereigniskanals zwischen Automaten sowie deren Zeitverhalten.

Ergänzend spezifiziert `execute` der Klasse `SMI` das Schaltverhalten eines einzelnen Automaten (s. Abb. 4.5(b)). Kurz zusammengefasst besteht dieses aus drei Phasen:

(1) Auswertung des `guards` des aktiven Zustands, um die Aktivierung einer Transition bei aktuellem Ereignis zu prüfen. Dieses ist ausgelagert in der `evalGuard`

[19] wenn wir annehmen, dass nur ein Eventbroker existiert, hätte der Test `self.clients->forAll(activeState.final)` ausgereicht

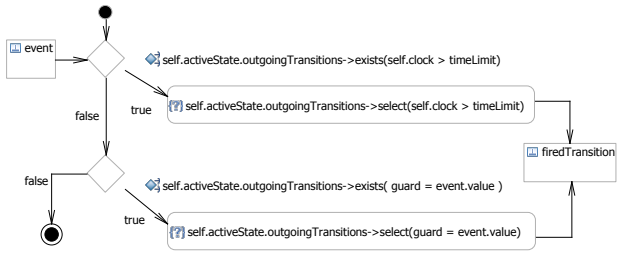


Abbildung 4.6: Transitionsaktivierung: StateMachineInstance#evalGuards

Operation und verbirgt sich hinter der Aktion `'self:self'`[20] Das aufzurufende Verhalten ist in Abb. 4.6 dargestellt. Sollte keine Transition aktiviert sein, finden keine weiteren Aktionen des Zustandsautomaten statt.

(2) Wird eine aktivierte Transition gefunden (d.h. `firedTransition` stellt eine Referenz auf diese Transition dar), wird das optionale *clock reset* ausgewertet und falls existent, die lokale Uhr `clock` zurückgesetzt.

(3) Als letztes wird die *action clause* der Transition ausgewertet, die zur Erzeugung eines neuen Ereignisses führen kann. Dieses erhält als `value` den String der an der Transition annotiert ist. Letztlich wird das neue Ereignis im Ereigniskanal für nachfolgende Verarbeitung abgelegt.

Zu Schritt (1) ist anzumerken, dass das Auswertungsverhalten des `guards` auf ein absolutes Minimum reduziert wurde, der für das Beispiel notwendig ist. Abbildung 4.6 zeigt das Verhalten der Transitionsaktivierung. Der erste Entscheidungsknoten überprüft, ob die lokale Uhr `clock` einen Wert größer als das in der Transition angegebene `timeLimit` hat. Der zweite überprüft durch Wertevergleich, ob das aktuelle Ereignis die Transition auslöst. Normalerweise müsste an dieser Stelle eine komplexere Logik eingebettet werden, die durch Auswerten von Ausdrücken die Aktivierungslogik modelliert. Zum Beispiel sind verschiedene Relationen zum Vergleichen von Uhrenwerten denkbar oder logische Operatoren, die das Eintreten von Ereignissen mit Uhrenwerten auf andere Weise als das hier stattfindende 'Oder' verbinden. Für unsere Fallstudie des Steam Boiler reicht diese reduzierte Variante der Aktivierung aus.

Als Konsequenz aus Schritt (3) folgt unmittelbar, dass die Ereignisliste um bis zu Anzahl von registrierten Zustandsautomaten in jedem Schritt wachsen kann. Da der Kanal ja wie beschrieben verlustfrei modelliert ist, führt dies u.U. zu langen Übertragungszeiten, je mehr Kommunikation aufkommt.

Mit diesen drei Operationen ist im Prinzip alles Notwendige spezifiziert, um die Zustandsautomaten aus Abb. 4.3 auszuführen. Zur Vollständigkeit fehlt lediglich die Instanziierung und Initialisierung der einzelnen Automaten, welche als `MActivity` modelliert und in Abb. 4.7 gezeigt ist.

4.4.2 Simulation des Beispiels

Das Beispiel des Steam Boiler als zwei Zustandsautomaten abstrahiert von der eigentlichen Wasserstandsmessung und enthält keinerlei Ereignisquellen für die Messwertereignisse `LevelMin`, `LevelNormal`, `LevelMax` sowie dem initiierenden Ereignis `SwitchOn` (vgl. Abb. 4.3). Damit ein Simulationslauf zustande kommt, benötigen

[20] Limitierung des Editors zeigt nicht das Ziel des Aufrufs (hier: `evalGuard`), nur eine mögliche Abfrage des `self`-Objekts

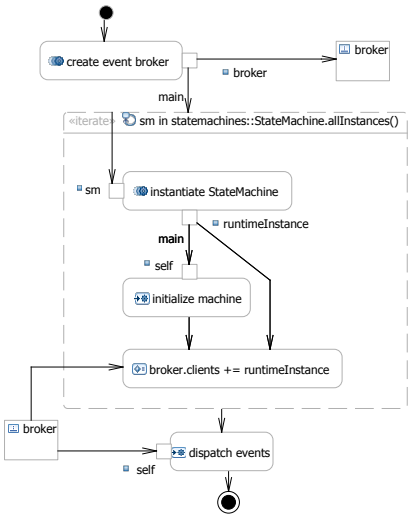


Abbildung 4.7: Initialisierung der SteamBoiler Zustandsautomaten

wir als Treiber einen TA, der diese Ereignisse emittiert. Für das Beispiel würde es ausreichen, eine Menge von TAs zu modellieren, die eine Kette von den genannten Ereignissen in verschiedenen Reihenfolgen emittieren.

Für unsere Experimente wurde ein anderer Weg beschritten und ein spezieller Automat implementiert, der sich ganz normal als `SMI` am Eventbroker registriert, aber in seiner `execute` Operation Ereignisse von der Kommandozeile bzw. aus einer Datei einlesen kann und diese in die Ereignisliste einhängt (vgl. Abschnitt 3.3.9). Dadurch ließen sich schnell verschiedenste Szenarien durchspielen und als Trace aufzeichnen. Im Allgemeinen können Simulationstreiber ergänzend zu den Ausdrucksmöglichkeiten einer Sprache eingebracht werden, um gewünschte Simulationsläufe oder Verhalten zu forcieren oder durch Bibliotheken zu ergänzen. So ist z.B. im Falle von Programmiersprachen, deren Semantik vollständig beschrieben wird, oftmals bereits eine Eingabe/Ausgabefunktion vorhanden und kann für diese Zwecke genutzt werden (vgl. z.B. C# im Anhang A).

Entscheidend für eine weitergehende Analyse ist die eigentliche Zustandsmodellierung in der Sprache TA. Fährt man eine Simulation im kanonischen Modus, so entsteht eine Zustandsfolge basierend auf den Aktionsdefinitionen, die den in Abschnitt 4.1.2 beschriebenen Mikrozuständen entspricht. Im Trace sind z.B. sämtliche Zwischenzustände der Operationen zur `eventQueue` enthalten, die Manipulation der Uhrenwerte in den Attributen `clock` über verschiedene Objektänderungen verteilt, etc. Um das intuitive Zustandsverständnis der Automatensprache TA herzustellen, nutzen wir den aggregierten Modus der Ausführung und *State Generating Transitions* (SGTs, vgl. Abschnitt 4.1.5).

Wir definieren einen Zustand in der Sprache TA genau dann für erreicht, wenn alle Automaten ein und dasselbe Ereignis abgearbeitet und ihre Uhren angepasst haben. Das heißt in der globalen Sicht schalten alle Automaten *gleichzeitig* und es existieren keine zusätzlichen Aktivierungszustände oder Transitionsübergänge. Zu diesem Zweck genügt es, wenn in der Hauptausführungsschleife die initiale und die letzte Transition als SGT markiert werden (vgl. Abb. 4.5(a)). Die erste SGT ist nötig, um alle vorher erzeugten und initialisierten Automaten als Anfangszustand festzuhalten. Jedes weitere abarbeiten eines Ereignisses führt zu einem neuen Zustand gdw. alle TAs auf dieses Ereignis reagiert haben.

An diesem Beispiel wird deutlich, wie diskrete Simulationszeit einer Sprache modelliert werden kann und sich in Relation zum Modellzustand verhält. Soll in einer Sprache stattdessen eine globale Uhr existieren, so müsste diese an einem zentralen Objekt verwaltet werden und bei Bedarf inkrementiert werden. Ähnlich verhält es sich mit (quasi) kontinuierlichen Größen, die zu bestimmten Zeitpunkten während der Simulation durch SGTs festgehalten („gemessen“) würden.

4.4.3 Analyse des Steam Boilers

In diesem Abschnitt werden wir eine Analyse des Beispiels vornehmen, um exemplarisch das allgemeine Vorgehen bei der Anwendung von LT-OCL zeigen. Zusätzlich reflektieren wir die Art der Zustandsmodellierung und Auswirkungen für die LT-OCL-Analyse.

Aus der Problemstellung des Steam Boiler aus Abschnitt 4.4.1 ergibt sich direkt die Sicherheitseigenschaft, das bei kritischem Wasserstand innerhalb von fünf Sekunden ein sicherer Zustand einzunehmen ist. Dieser kann entweder das zurückführen des Wasserpegels in erlaubte Toleranzgrenzen sein (kenntlich durch das `LevelNormal` Ereignis) oder der Nothalt des Systems (Zustand `EmergencyStop`). In LT-OCL kann dieser Sachverhalt als always/eventually Kombination über die Zustände der Zustandsautomaten wie folgt ausgedrückt werden:

```
let cp : SMI = SMI.allInstances()->select( metaObject.name='ControllerProgram' ),
  pump : SMI = SMI.allInstances()->select( metaObject.name='Pump' )
 in
always (cp.activeState.name='Regulating' implies (cp.clock <= 5 and
 eventually ((cp.activeState.name='Normal' and pump.activeState.name='On') or
   (cp.activeState.name='EmergencyStop'and pump.activeState.name='Stop'))))
```

Im `let`-Teil des Ausdrucks werden die beiden TAs per Namen selektiert. Dazu hilft die logische *instanceOf*-Beziehung, da durch `metaObject` einfach zur Automatendefinition navigiert und der Name verglichen werden kann. Besonderes Augenmerk ist auf die zeitliche Auswertung der Teilausdrücke zu legen: sollte `cp.activeState.name='Regulating'` innerhalb des *always*-Operators wahr werden, dann werden mit allen Ausdrücken über `cp` innerhalb von `eventually` Attributwerte desselben Objekts zu späteren Zeitpunkten überprüft. Im Allgemeinen heißt das, dass bei verschachtelten temporalen Operatoren Objektbindungen durch äußere Operatoren möglich sind, um zukunftsbezogene Aussagen zu Eigenschaften dieser Objekte auszudrücken.

Es sei angemerkt, dass der gesamte LT-OCL-Ausdruck für alle Ausführungsläufe $Undefined$ ist, bei denen der let-Ausdruck für *cp* and *pump* ungebunden bleibt (vgl. Abschnitt 4.3.3). Dieser Teil des Ausdrucks ist in der Regel immer der erste Schritt zu einer modellspezifischen LT-OCL-Bedingung, es sei denn, temporale Invarianten für alle Instanzen einer Klasse (oder sogar auf Sprachebene) sollen spezifiziert werden. Ein Beispiel hierfür könnte sein, dass alle Automaten immer einen Endzustand erreichen:

```
context StateMachineInstance inv:
eventually (self.activeState.final)
```

Folgen wir dem Beispiel weiter lässt sich z.B. für das kritische Unterschreiten des Wasserstandes (`LevelMin`-Ereignis) die Bedingung formulieren, dass das Kontrollprogramm in jedem Fall die Pumpe aktiviert:

```
always (Event.allInstances()->forAll( value='LevelMin' implies
    eventually (pump.activeState.name='Pumping')))
```

Dadurch kann abgesichert werden, dass unser Ereigniskanal kein Ereignis verschluckt und das vom Kontrollprogramm gesendete `OpenPump`-Ereignis garantiert ankommt. Zudem wird deutlich, dass ein temporaler Ausdruck nicht immer über einen let-Ausdruck einsteigen muss, sondern beliebige Objekte als Referenz herausgreifen kann.

Interessanter wird es, wenn wir versuchen etwas über konkrete Zeitpunkte bzw. zeitliche Relationen der Ereignisse auszudrücken. Zum Beispiel könnte man fordern, dass die Pumpe zeitlich immer spätestens nach einem Uhrenschlag anfängt zu pumpen, d.h. in den Zustand `Pumping` übergeht. Mit Hilfe des `next`-Operators lässt sich dies wie folgt beschreiben:

```
always(cp.activeState.name='Regulating'
  implies next(pump.activeState.name='Pumping'))
```

Aus temporaler Sicht ist diese Forderung intuitiv und würde konsequent der Anforderung genügen, möglichst schnell die Pumpe zu aktivieren. Beachtet man hingegen die operationale Semantik unserer Sprache TA, wird deutlich, dass diese Bedingung in einigen Fällen nicht eingehalten werden kann, da sich der Schaltvorgang durch mehrere Ereignisse in der `eventQueue` verzögern kann. Befinden sich z.B. durch weitere Zustandsautomaten (wie dem Simulationstreiber) zu einem Zeitpunkt zusätzlich zu einem `LevelMin` weitere Ereignisse in der Liste, dann wird das Kontrollprogramm sein `OpenPump`-Ereignis nur ans Ende einreihen können und die Pumpe erst nach Abarbeiten der anderen Ereignisse schalten.

An dem `next`-Operator sieht man deutlich, wie zwei Zeitpunkte relativ zueinander adressiert werden können. Im Gegensatz dazu besteht bei dieser Sprachdefinition eine weitere Möglichkeit darin, durch expliziten Zeitzugriff, vorausgesetzt natürlich eine Zeitvariable ist vorhanden. Zum Beispiel könnte man versuchen, die vorangegangene Bedingung über das `clock`-Attribut beider Automaten zu reformulieren. Das Problem hierbei ist allerdings, dass die Pumpe ihre lokale Uhr niemals zurücksetzt und somit ein relativer Vergleich der Uhren unmöglich wird.[21] Man könnte sich behelfen, in dem man den Pumpenautomat durch Zurücksetzen der Uhren abwandelt, z.B. durch ändern des Transitionslabels bei `On -> Pumping` zu `OpenPump/clock := 0`. Dadurch wäre folgende Bedingung möglich:

```
always(cp.activeState.name='Regulating'and pump.activeState.name='Pumping'
    implies pump.clock <= cp.clock + 1)
```

Dies führt uns zur Diskussion über die Relation von Zeitgrößen, Zuständen und temporalen Operatoren. Spezifisch an diesem Beispiel ist die Koinzidenz von adressierten Zeitpunkten durch temporale Operatoren und dem Modellzustand. Letztlich ermöglicht die Adressierung des Zustandes durch `activeState` in Kombination mit den temporalen Operatoren nur genau dann eine intuitive Beschreibung, wenn der tatsächliche Zustand im Trace exakt derselbe ist. Mit anderen Worten, im kanonischen Modus der Ausführung würden diese LT-OCL-Bedingungen in den meisten Fällen fehlschlagen. Wenn zum Beispiel überhaupt kein Uhrenreset stattfinden würde, wären aus globaler Zustandssicht die Uhren niemals synchronisiert und die folgende Bedingung immer falsch:

```
context StateMachineInstance inv:
always (SMI.allInstances()->forAll( sm2 | self.clock = sm2.clock ))
```

[21] Dies wäre nur denkbar, wenn wir über LT-OCL explizit Zugriff auf alle Zustände im Trace hätten

Dies liegt an den Mikrozuständen der kanonischen Ausführung, bei der die Uhrenwerte über verschiedene Mikrozustände verteilt inkrementiert werden. Die Zustandsbildung stellt demnach eine wesentliche Voraussetzung und Herausforderung für sämtliche Analysen dar, wie auch die Fallstudie für C# zeigt (vgl. Anhang A).

4.5 Umsetzung der Konzepte

Die vorgestellten Metamodelle und Konzepte des εMOF mit logischer Instanziierung und MActions wurden im Rahmen dieser Arbeit experimentell implementiert (vgl. [127]). Als Basis der Java-Implementierung diente die eclipse-Plattform mit den Modellierungsprojekten EMF [128], OCL [129] und GMF [114]. Als Editor für Aktionsflüsse wurde ein GMF-generierter und am eclipse-UML2-Projekt angelehnter, graphischer Editor konzipiert, dem auch die in dieser Arbeit gezeigten Screenshots direkt entnommen wurden. Zur Umsetzung der Laufzeitumgebung wurde ein Interpreter realisiert, der mittels der dynamischen Objektimplementierung reflektiv EMF-Metamodelle (ecore-Dateien) und Instanzen (XMI-Dateien) lädt und entsprechend der Aktionssemantik verarbeitet. Dazu konnte die für EMF existierende OCL-Implementierung übernommen und angepasst werden. Die Vorgehensweise entsprach der in Abschnitt 2.6 beschriebenen Einbettung der OCL-Auswertungsumgebung in den MActions-Interpreter auf Implementierungsebene. Eine Fortführung des Frameworks als offizielles und qualitätsgesichertes Open-Source-Projekt *Model Execution Framework* (MXF) wurde initiiert (vgl. [130]).

Für den gesamten Ansatz wurde keine detaillierte Laufzeitmessung oder Ressourcenverbrauchsanalyse durchgeführt, dennoch konnten stichprobenartig Ausführungen mit einer operational äquivalenten Java-Implementierung verglichen werden, um zumindest Grundtendenzen gegenüber einer direkten Simulatorimplementierung/eines generativen Ansatzes aufzuzeigen. Dabei stellte sich heraus, dass für die reine Interpretation von OCL-Abfragen mindestens eine ca. 30-35% längere Laufzeit zur Interpretation angenommen werden muss.[22] Auch wenn diese vergleichsweise gute Performance für einzelne Aktionen überzeugt, akkumuliert sich der Overhead des Laufzeitmodells mit Verwaltung von Plätzen, zusammensetzen der OCL-Umgebung, Navigation von Transitionen etc. auf ein Vielfaches. So lief z.B. das in Anhang A gezeigte *BubbleSort*-Programm im Interpreter um ein Mehrhundertfaches langsamer ab als eine vergleichbare Java-Implementierung.[23]

Für die Experimente zur Temporallogik LT-OCL konnte ebenfalls die bestehende EMF-OCL-Implementierung adaptiert werden. Anstatt die OCL-Grammatik (und somit den Parser) formal wie in Kapitel 4.3.2 beschrieben zu erweitern, konnten die Temporaloperatoren als *user defined*-Operationen eingebunden und in einer erweiterten Auswertungsumgebung behandelt werden. Im Prinzip wurde dabei jede Temporalformel ähnlich zum *Imperative Future*-Ansatz von Barringer et al. [131] als Menge von „aktiven Objekten“ gehandhabt, welche je nach Variablenbindung durch Iteratoren partiell bei jeder Zustandsänderung überprüft wurden. Als Basis der Experimente diente das Trace-Metamodell aus Abschnitt 4.1.3 und größtenteils das in 4.4.1 gezeigte Beispiel-Metamodell für Zustandsautomaten. Performancemessungen wurden für diesen Teil nicht unternommen.

[22] Die Zeit zum Parsen von OCL-Ausdrücken ist hier nicht eingerechnet, da der Parser nur zum Zeitpunkt der Interpreterinitialisierung einmalig ausgeführt wird.

[23] Verglichen wurden nur kleine Arrays im zehnstelligen Bereich, da zudem der Speicherverbrauch aufgrund der nicht optimierten Modell- und Variablenstrukturen beim Zigfachen lag.

KAPITEL 5

Diskussion

Dieses Kapitel diskutiert die in Kapitel 2 bis 4 dargestellten Konzepte und setzt sie in Relation zu verwandten Arbeiten. Neben einer generellen Diskussion zur Formalisierung und zum Ansatz der operationalen Semantik mittels MActions in 5.1 reflektiert Abschnitt 5.4 die Vorgehensweise dynamischer Modellanalysen im Kontext von ausführbaren Modellen.

5.1 Motivation und Hintergründe zu MActions

Die formale Definition von Programmiersprachen, ob denotationell, axiomatisch oder operational, behandelt im Kern das gleiche Problem: der Syntax unmittelbar eine (Ausführungs-)Semantik zu verleihen. Den frühen Arbeiten zur später 'denotationell' genannten Semantikdefinition von McCarthy, Scott und Strachey in der 60er und 70er Jahren ist gemein, dass der Syntax durch Abbildung auf ein mathematisches Objekt die Semantik ebendieses Objekts zugeordnet wird (vgl. [86], [84]). Die Idee einer Übersetzungsfunktion der Form $[\![syntax]\!] \rightsquigarrow object$ von überwiegend textuellen Sprachen (Algol60, LISP, ML) erhielt daraufhin Einzug in die Arbeiten von Plotkin, Milner und anderen. Allerdings unterschied sich der operationale Ansatz dadurch, dass nicht erst ein Programm *vollständig* übersetzt wird (z.B. auf ein Funktionskalkül), sondern die Auswertung Schrittweise als Abarbeitung mittels eines mathematischen Objekts/Maschine *direkt*[1] beschrieben wird, also der Form $\langle AS, \sigma \rangle_{\text{MACHINE}}$. Dies ging einher mit der nun betrachteten abstrakten Syntax (AS) und den in Kapitel 2.5 beschriebenen SOS-Schlussregeln über Konfigurationen σ mit Variablen und Mengen zur Zustandsrepräsentation. Das diese 'abstrakten Maschinen' bis dato ungenügend in der Handhabung waren, beschreibt Gurevich als Motivation für den ASM-Kalkül (vgl. [97]):

> Computation models and specification methods seem to be worlds apart. The evolving algebra project started as an attempt to bridge the gap by improving on Turing's thesis[...]. We sought more versatile machines which would be able to simulate arbitrary algorithms in a direct and essentially coding-free way. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and

[1] Vgl. Motivation bei Plotkin in [21]

> real-time protocols, etc.. The simulator is not supposed to implement the algorithm on a lower abstraction level; the simulation should be performed on the natural abstraction level of the algorithm.

Das demnach ASM nicht primär für die Beschreibung operationaler Semantik entwickelt wurde, sondern als allgemeines formales Spezifikationsinstrument, äußert sich insbesondere an der fehlenden Bindung zur (abstrakten) Syntax einer Sprache in oben beschriebenem Sinne. Als erster Schritt zur Definition von ASM-Regeln muss deshalb die (abstrakte) Syntax in eine Signatur und Algebra übersetzt werden, auf der dann Regeln Anwendung finden, d.h. $AS \rightsquigarrow \langle \Sigma, \mathbb{A} \rangle_{\mathbf{EA}}$. Für den Übergang '$\rightsquigarrow$' existiert zunächst kein einheitliches Vorgehen, sondern es werden meist Notationserweiterungen oder andere Hilfsmittel eingeführt, um Sätze einer Sprache 'greifbar' zu machen. Beispiele hierfür sind der Zugriff auf Notationen durch abstrakte (informell beschriebene) Funktionen [24][27][132][133][98], eine Verbindung mit dem AST eines Parsers durch Termersetzung [26] oder den in dieser Arbeit gegangenen Weg über eigene explizite Definition (vgl. 2.6.2).

Folgende Tatsache war neben dem Pragmatismus (s. Abschnitt 5.1.1) eine Hauptmotivation für die MActions: die abstrakte Syntax liegt bereits als MOF-Metamodell vor und sollte möglichst *direkt*[2] durch eine abstrakte Maschine interpretiert werden und einen eigenen Kalkül bilden. Diese Form der direkten Modellinterpretation ist hier in doppeltem Sinne zu Verstehen. Erstens soll die abstrakte Syntax nicht durch Übersetzung in ein mathematisches Objekt oder andere Form von Modell reformalisiert werden, d.h. das Metamodell wird direkt interpretiert: $\langle \mathcal{M}, \mathcal{M} \rangle_{\text{MActions}}$. Zweitens sollen Metaklassen, die ein Konstrukt der Sprache strukturell beschreiben, möglichst unmittelbar um eine operationale Definition erweitert werden können. Jedes einzelne Konstrukt und Konzept einer Sprache soll möglichst lokal an den betreffenden Metaklassen definiert werden können.

Vor dem Hintergrund, dass keine Ausführungssemantik für MOF existiert[3] und jedes Metamodell bislang informell dynamische Semantik einzelner Klassen definiert, müssten also entweder etablierte Kalküle in geeigneter Form adaptiert oder ein neues Regelwerk definiert werden. Zu Beginn stellten sich drei Kernprobleme in Hinblick auf eine operationale Semantik:

1. Metamodelle definieren die abstrakte Syntax einer Sprache als objektorientierte Modelle. Diese OO-Formalisierung liegt primär in graphischer Notation vor (UML-Klassendiagrammen) und ist nicht über universelle Algebren oder andere mathematische Strukturen beschrieben (vgl. [8] [3]).
2. Instanzen eines Metamodells – die eigentlichen Modelle – besitzen laut MOF keine standardisierte Notation[4]. MOF enthält zwar das beschriebene Instanzmodell in Anlehnung an die UML-Infrastruktur (vgl. Abschn. 2.3), allerdings gibt es keine offizielle MOF-Objektnotation.
3. Die existierenden Kalküle zur operationalen Semantik sind nicht über ein Metamodellierungsframework definiert. Weder SOS noch ASM nutzen Objektstrukturen (Attribute, Assoziationen, Vererbung etc.) als Konfigurationen und

[2] Ganz im Sinne Plotkins

[3] Operationen in Metamodellen haben keine formale Semantik, für andere Formen von Semantikdefinitionen existierten nur Forschungsansätze, s. Abschn. 5.3. Die Situation ist vergleichbar mit der Ausgangslage von ASM, nur dass keine Algebren, sondern Objektgraphen vorlagen

[4] Ausgenommen sind XMI Repräsentationen nach [78]. XMI als textuelle Repräsentation von Metamodellen wurde aus Gründen der Handhabbarkeit und Leserlichkeit als unpraktikabel empfunden und verworfen

sind in der Lage, reflektiv solche Strukturen zu navigieren (s. Laufzeitmodell Abschn. 2.6.1).

Die UML-Notation von Metamodellen mag hinderlich erscheinen, aber natürlich ist die graphische Modellierungen seit Anbeginn der theoretischen Informatik weit verbreitet (Graphen, Automaten, Petri-Netze) sind stets durch mathematische Objekte wie Graphgrammatiken (textuell) definiert[5]. Aus Kapitel 2.4 wird deutlich, dass letztlich der für UML existierende Ansatz von Richters et al. adaptiert wurde, um Metamodelle auf Algebren abzubilden, nicht zuletzt um keine erneute Formalisierung von OCL vorzunehmen (vgl. [85] sowie OCL Standard [17], Appendix A).

Die Frage, die sich primär stellt, ist also nicht, ob und wie sich Metamodelle durch algebraische Strukturen definieren lassen, sondern ob eine Verbindung mit den operationalen Kalkülen SOS oder ASM im Hinblick auf Punkt 3 sinnvoll ist und wie Transitions- bzw. Updates im Hinblick auf Objektstrukturen aussehen könnten (Punkt 2). Wie in Kapitel 2.6.1 gezeigt, sind diese Strukturen zwar umsetzbar aber unserer Auffassung nach im Kontext von OO-Metamodellen nicht praktikabel. Deshalb stellt diese mathematische Formalisierung nur eine sekundäre Definition der MActions dar, die eigentliche Definition gibt Kapitel 3.

Aus dieser Argumentation heraus bot es sich an, eine objektorientierte Form von Transitions- bzw. Updateregeln zu konzipieren, die sich harmonisch in das MOF-Framework einfügt und die Beobachtungen zur logischen Instanziierung berücksichtigt (s. Abschn. 2.3.1). Im Hinblick auf OCL als prädikatenlogische Sprache lag eine Erweiterung ähnlich der ASM-Updates nahe[6], wobei die Zuordnung von Regeln zu Metaklassen eine Kapselung einzelner Semantikbausteine ermöglicht. Mittels Operationen lassen sich diese Regeln entweder in Metaklassen einbinden oder 'nebenher' als Aktivitäten definieren. Polymorphie und dynamisches Binden erlauben vielfältige Möglichkeiten das Verhalten zu spezifizieren und zu strukturieren (z.B. mit Entwurfsmustern [112], s.a. Abschn. 3.2.4). Dynamische Funktionen in ASM bilden ein ähnliches Pendant, um über Updates anderes Verhalten einer Funktion zu erreichen (z.B. durch Simulation von dynamischer Bindung), allerdings nicht in typisierter Form mit klarer Schnittstellenveränderung (Ko-/Kontravarianz), die bereits durch die Typeigenschaft von Klassen vorgegeben ist. Nicht zuletzt aufgrund der Parallelitätseigenschaften (s.Abschn. 2.5.3) und zustandsbasierten Herangehensweise (s.u.) wurde der Weg der Aktionssemantik beschritten.

Die graphische Syntax der MActions, welche aus UML[7] entlehnt ist, stellt eine pragmatische Möglichkeit dar, komplexe Zusammenhänge des operationalen Flusses mit einer etablierten Syntax zu notieren. Tatsächlich wurde die Idee einer graphischen Notation bereits von Plotkin in [21] vermerkt[8], wo verschiedene „Datenflüsse“ graphisch visualisiert sind (allerdings ohne präzise Semantik). Eine Diskussion, ob eine graphische Darstellung einer textuellen vorzuziehen ist, soll an dieser Stelle vermieden werden[9]. Es soll aber darauf hingewiesen werden, dass graphische Nota-

[5] s. z.B. [134]

[6] OCL verhält sich analog zu den prädikatenlogischen Formeln des ASM-Kalküls, vgl. Abschn. 2.5.1 und 3.2.1

[7] Nur die Syntax wurde teilweise übernommen und adaptiert, s.a. 3.1

[8] Er spricht allerdings von 'pictures' oder 'flowcharts' um Regeln zu visualisieren; In seinem 2004 erschienen Rückblick [87] zu den Ursprüngen operationaler Semantik wird dieser Gedanke ebenfalls aufgegriffen

[9] Generell stellt die Debatte, ob graphische oder textuelle Notationen ≫besser≪ geeignet seien eine immer wiederkehrende Schimäre dar: für beides gibt es Vor- und Nachteile, abhängig vom Anwendungskontext sowie persönlichen Vorlieben von Nutzern. Für einen Vergleich von Aktivitäten in textueller und graphischer Syntax sei z.B. auf [135] verwiesen.

tionen nicht weniger formal sind – sie lassen sich z.B. mittels Graphgrammatiken formalisieren (s.a. GenGed [16]).

5.1.1 Einordnung: Pragmatismus und Formalismus

Dieser Abschnitt ordnet MActions in Relation zu SOS und ASM ein und zeigt die Ausdrucksmöglichkeiten und Grenzen auf. Insbesondere der Pragmatismus der Herangehensweise soll formalen Beweiskalkülen gegenübergestellt und diskutiert werden.

Ein formales System von Inferenzregeln zur Definition einer Sprache bietet eine rigorose Semantik, auf Basis dessen formale Beweise erstellt werden können, z.B. Beweise der Typsicherheit oder Korrektheit von Programmen. Für operationale Kalküle gilt die Eigenschaft der Beweisbarkeit aus mehreren Gründen nur mit Einschränkungen, was überwiegend auf die Art der Schlussregeln zurückzuführen ist. Blickt man auf die SOS-Regeln (vgl. Def. 10[10]), so wird deutlich, dass Beweise für komplexere Sprachen insb. wegen den zustandsbehafteten Konfigurationen sehr schnell an Komplexität gewinnen, dasselbe gilt für ASM. Eine vollständige Diskussion dieses Themenkomplexes kann an dieser Stelle nicht geführt werden, aber wir teilen diese Auffassung mit anderen Autoren[11] und sehen den Hauptzweck der operationalen Semantik in einer abstrakten, ausführbaren, für Menschen leicht verständlichen und unzweideutigen Beschreibung der Ausführungsschritte einer Sprache. Nichttriviale Beweise mittels struktureller Induktion über die abstrakte Syntax oder Ableitungsbäumen sind für komplexere Sprachen oftmals leichter durch eine Reformalisierung in dafür geeigneteren Kalkülen und unter Einsatz von Theorembeweisern zu führen. Operationale Semantiken spezifizieren dafür ein Referenzverhalten, das für Analysen, Implementierungen und Verhaltensvergleiche herangezogen werden kann (s.a. Kapitel 4).

Dennoch unterscheidet sich MActions im Grade der Ausdrucksmöglichkeit von SOS oder ASM, da die zu Grunde liegenden Meta-Metamodelle verschieden sind. Auch wenn die Metamodellierungsarchitektur MOF vermeintlich meta-zirkulär *in sich* abgeschlossen ist, fußt sie natürlich auf Konzepten der Mathematik (s. Abb. 5.1). Man könnte hier auch von Instanziierung sprechen, allerdings in einem meta-mathematischen Sinne (nicht zu vergleichen mit den Instanziierungsbeziehungen der Objektsprache aus Kapitel 2.3). Algebren und formale Systeme als Teilgebiete der Ma-

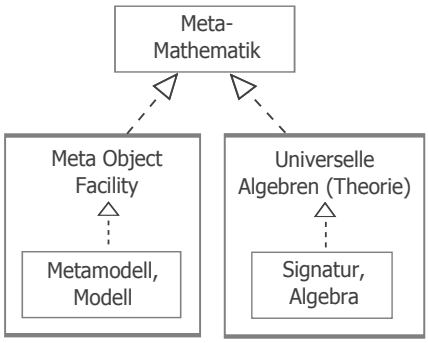


Abbildung 5.1: Meta-Theorie: Einordnung von Algebren und MOF Metamodellierungsarchitektur

thematik können in ebendieser Weise aufgefasst werden, jedoch sind die Grenzen für

[10]Gleiches gilt für GSOS-Systeme, nur hier sind Konfigurationen als Terme und Variablen beschrieben

[11]z.B. Plotkin [87], Mosses [136], Prinz [26], Scheidgen [71]

MOF wesentlich enger, was die Mächtigkeit des Formalismus deutlich einschränkt. Als Folge bieten operationale Semantiken, die wie SOS und ASM auf mathematischen Kalkülen fußen, im Vergleich zu MOF und MACTIONS durch Ihre Erweiterbarkeit mächtige Abstraktionsmöglichkeiten. Zum Beispiel lassen sich Differentialfunktionen, Zufallsfunktionen oder zeitkontinuierliche Echtzeitfunktionen leicht *per Definition* in eine Signatur/Algebra einführen, von der zunächst nur abstrakt Eigenschaften wie *„eine Operation muss eine (partielle) Abbildung sein“* gefordert werden (vgl. [26]). Mithin ist zwar die *Interpretation* des ganzen Kalküls immer noch gültig und schlüssig[12], letztlich würde eine Implementierung aber mit solchen Erweiterungen Schwierigkeiten haben. Auch wenn diese Beobachtung gänzlich für den Übergang von (beinahe allen) mathematischen Modellen zu einer Implementierung auf einem Computer zutreffend ist[13], so spielt es in der Praxis eine entscheidende Rolle z.B. bei der Entwicklung eines Simulators für eine neue DSL.

Unter pragmatischen Gesichtspunkten und im Hinblick auf Implementierungen wurde bei MACTIONS die Strategie verfolgt, mittels der OPAQUEACTION eine Schnittstelle aufzunehmen, die unter gewissen Rahmenbedingungen die Aktionssemantik um allgemeine Funktionen und Modelltransformationen *erweitern*[14] kann. Dadurch lassen sich z.B. komplexere Berechnungen und Algorithmen als Bibliotheken bereitstellen, deren Modellierung mittels Aktionen sehr umfangreich sein würde. Dies gilt insb. im Hinblick auf eine schnelle Entwicklung einer Simulationsimplementierung. Semantisch kann man argumentieren, dass diese Art von 'black-box' sich außerhalb des Metamodells/der Aktionssemantik bewegt, sich aber harmonisch als *good citizen* in den Kalkül einfügt. Da Aktionspins eine präzise (Funktions-)Signatur und Übergabesemantik definieren, entspricht diese Erweiterung ungefähr der Möglichkeit in ASM eine Signatur frei durch eine Algebra festzulegen[15]. Im Gegensatz dazu wird aber die Trennlinie durch die OPAQUEACTION enger geführt als bei ASM, so dass die Kernsemantik in jedem Fall minimal bleibt und ausschließlich durch diese Form der Bibliotheken erweitert werden sollte. Experimentell konnte z.B. eine Aktionsimplementierung QVTACTION eingefügt werden, die einen Zeichenkettenparameter über einen Eingabepin als eigenständige Transformation ausführte[16]. Spätestens durch solche beliebigen Manipulationsmöglichkeiten am Laufzeitmodell wird deutlich, das die Beweismöglichkeiten über Aktionsdefinitionen gering sind.

Sieht man von diesen metatheoretischen Aspekten ab, zeigen sich deutliche Gemeinsamkeiten in vielen Aspekten der Kalküle, die Tabelle 5.1.1 zusammenfasst. Für Details zu den einzelnen Einträgen sei auf die Kapitel 2.5 und 3.1 verwiesen. Ersichtlich ist, dass neben den verschiedenen Formalisierungen für abstrakte Syntax, Transitionsregeln und Zustand die Hauptunterschiede in den Strukturierungsmitteln, dem Umgang mit Parallelität sowie dem generellen Zustandsverständnis liegen. Bevor wir die Unterschiede zur Ausführungssemantik ausführlicher in Abschnitt 5.2 analysieren, sollen die Strukturierungsmittel beleuchtet werden, die dessen Basis darstellen.

[12] Ggf. eine erweiterte Interpretation, Widerspruchsfreiheit vorausgesetzt

[13] Sobald es um Unendlichkeit von z.B. Mengen im Kalkül geht, steht dem eine faktische Endlichkeit in der Implementierung entgegen

[14] 'erweitern' ist hier nicht im Sinne von Berechenbarkeit gemeint. Die Turing-Vollständigkeit der MACTIONS wurde in Abschn. 3.5 bewiesen.

[15] Auch vergleichbar mit **shared**- und **monitored**-Funktionsdefinitionen in ASM

[16] Basierend auf der *medini QVT Engine*, s. [137]

	SOS/GSOS	ASM	MActions
Abstrakte Syntax	Σ-Terme	Σ-Terme	Metamodelle
Transitionsregeln	Schlussregeln, Termersetzung	Updates, Logik erster Stufe	Aktionen, Kontrollfluss, Logik erster Stufe (OCL)
Strukturierungsmittel	—	Regelmakros, Funktionen	Klassen, Attribute, Aktivitäten, Operationen, Vererbung, Polymorphie
Maschinenparadigma	LTS	Zustandsautomat	Stapelmaschine
Zustandsgrößen	Variablen, Σ-Terme	Σ-Algebren	Metamodellinstanzen
Zustandsübergänge	Ableitungssequenzen und -bäume	Δ-Update Mengen	Modus abhängig (kanonisch, aggregiert)
Parallelität	(implizit in Regeln)	Agenten, partielle Abläufe	Threads, Aktionsraum, linearisierte Abläufe
Nichtdeterminismus	Alternative Regeln	**choose**-Konstruktor	fork- und Decision-Knoten
Zeit	—	diskret, linear (partielle Halbordnung)	diskret, linear (totale Halbordnung)

Tabelle 5.1: Vergleich der Kalküle SOS, ASM und MActions

5.1.2 Strukturierung von Semantikdefinitionen

Strukturierungsmitteln in operationalen Kalkülen kommt bei der Definition nichttrivialer Sprachen eine besondere Bedeutung zu. Schaut man sich Metamodelle von Programmier- und Modellierungssprachen mit teilweise hunderten Metaklassen an, welche die abstrakte Syntax einzelner Konstrukte widerspiegeln, so stellen sich schnell Fragen nach der Beherrschbarkeit algebraischer Definitionen und deren Schlussregeln. Nimmt man alleine das Beispiel C#, so müssten in SOS geschätzt fünfzig bis hundert Schlussregeln aufgestellt werden, um die Sprache in allen Facetten zu beschreiben (vgl. Appendix A). Peter D. Mosses leitete daraus u.a. seine Motivation zur Aktionssemantik ab und formulierte in *Theory and Practice of Action Semantics* einige Nachteile, die – wenn auch ursprünglich auf denotationelle Definitionen zielten – für algebraische Kalküle ebenfalls gelten (vgl. [136])[17]:

> 1. The definitions of semantic domains are globally visible throughout a denotational description. [...] Changes in domain definitions are often required when extending the described language with further constructs — e.g., a change from the direct style to the continuation style when adding jumps, or to power domains when adding nondeterminism. [...]
> 2. The way λ-notation is used for specifying semantic entities depends strongly on the details of domain definitions. [...] Also, the notation used for constructing elements of sum domains $D_1 + \cdots + D_n$ from the summands, and for case selection on such elements, is sensitive to the positions of the D_i in the sum: inserting a new summand in

[17] Mosses kommt allerdings zu einer ganz anderen Lösung in „seiner“ Aktionssemantik, die pragmatisch in formalisiertem Englisch operationale Semantik definiert

the middle can involve major changes throughout, just to preserve well-formedness of the notation.

3. All programming concepts have to be reduced to pure functions. This corresponds to translating arbitrary programs into a (lazy) functional programming language, and the amount of encoding required can be large.

Zusammengefasst leiten sich daraus allgemeine Forderungen nach Modularisierung (*information hiding* und Kapselung), Wiederverwendung, Kompositionsmechanismen und der Möglichkeit zustandsbasiert zu arbeiten ab, die durch objektorientierte Sprachmittel in großen Teilen gegeben sind. Dies ist ein starkes Argument für die Definition von OO-Metamodellen und der Nutzung von Objektstrukturen, bei denen Transitionsregeln innerhalb von Operationen direkt spezifiziert sind (s. Abschn. 3.2).

Blickt man im Vergleich auf ASM-Beschreibungen, die mit Strukturierungsmitteln der Regelmakros, Regel-Aufruf und **let**-Konstrukte ebenfalls diese Problematik adressieren, fällt als Unterschied insbesondere der *prozedurale* Charakter der Beschreibung ins Auge: Regeln und Domänen sind immer global definiert, bedingte Regeln werden mittels globaler Prädikate auf Kontexte der abstrakten Syntax eingeschränkt, et cetera. Dies setzt sich in der Modellierung von Zustandsgrößen fort, bei der nicht alle zusammenhängenden Größen durch eine einzelne Menge repräsentiert werden, sondern immer durch mehrere. Um zum Beispiel einen Agenten mit einem Laufzeitmodus zu versehen, bedarf es einer Funktion der Form $mode : Agent \rightarrow$ Mode, um ihm einen Stapel zuzuordnen einer weiteren Funktion $stack : Agent \rightarrow Seq($StackElement$)$. Verallgemeinert muss also für jedes Attribut des ≪Typs≫ *Agent* eine neue Funktion eingeführt werden (s.a. Kapitel 2.6.1)[18]. Der Unterschied der Laufzeitgrößen ist somit strukturell bedingt. Konfigurationen und abstrakte Syntax bilden eine gemeinsame Signatur.

Im direkten Vergleich entspricht der Übergang von ASM zu MActions somit bezüglich der Strukturierung dem Übergang von einer prozeduralen Sprache hin zu einer objektorientierten. Gurevich greift diesen Gedanken in seinem Asml Projekt ebenfalls auf und kommt zur Erweiterung von ASM um Klassenkonzepte (vgl. [30]). Börger et al. definiert zu diesem Problem *Turbo ASMs* in [111], mit sequentieller Komposition und rekursiven Untermaschinen, die lokale Zustände verwalten. Aufgrund der verschiedenen Erweiterungen in den letzten Jahren ist eine weitergehende Differenzierung im Vergleich schwierig und für sich genommen eine Forschungsaufgabe, die wir hier nicht leisten können. An dieser Stelle sei auf die Errungenschaften von Gurevich, Rozenberg, Börger et al. hingewiesen, für eine Übersicht siehe u.a.[138][104][139][140].

5.2 Ausführungssemantik

Die drei Kalküle SOS, ASM und MActions sind unzweifelhaft gleich mächtig. Auch wenn einige Unterschiede existieren, wie z.B. das Rekursion nicht im ASM-Basismodell enthalten ist (vgl. [141]), bieten alle drei einen Regelsatz für Sprachdefinitionen, die turingvollständig sein können. Die Hauptunterschiede zwischen den Kalkülen bestehen in erster Linie in der grundsätzlichen Modellierung der Konfigurationsänderungen der zugrundeliegenden abstrakten Maschinen (vgl. Tabelle 5.1.1).

[18] Für alternative Abbildungen von Objekten verhält es sich ähnlich. Selbst wenn eine einzelne Domäne z.B. Attributwerte als Tupel repräsentiert, so müssten für den Zugriff auf Attributwerte Projektionsfunktionen angegeben werden.

Dies äußert sich in vielschichtigen Aspekten der Transitionsregeln, dem Zustandsverständnis und der Parallelität, so dass auf unterschiedliche Weise Äquivalentes erreicht wird. Die Gemeinsamkeiten und Unterschiede der drei Ansätze werden im Folgenden anhand einer Beispielsprache verdeutlicht und diskutiert.

Eine simple Programmiersprache, die arithmetische Ausdrücke, Zuweisungen, if-then-else sowie while-Schleifen umfasst, sei gegeben durch zwei Produktionsregeln der abstrakten Syntax (adaptiert von [21]):

$L = c$ mit:

$$e ::= m \mid v \mid (e + e') \mid (e - e') \mid (e * e') \tag{5.1}$$

$$c ::= \varepsilon \mid c := e \mid c;\ c' \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \mid \textbf{while } e \textbf{ do } c \tag{5.2}$$

Die Sprache L repräsentiert die Klasse der WHILE-Sprachen, also einfache Berechnungen mit Variablen, bei der Anweisungen sequentiell abgearbeitet werden. Zum Beispiel würde die Fakultätsfunktion durch das Programm:

$$y := 1; \textbf{while } x \textbf{ do } y := y * x; x := x - 1 \tag{5.3}$$

beschrieben. Bedingungen werden wie in der Programmiersprache C als *true* gewertet, wenn ein Wert von Null verschieden ist. Da es um die abstrakte Syntax geht, werden Klammerungen bei Anweisungen und Ausdrücken nur genutzt, um Eindeutigkeit herzustellen, zum Beispiel:

$$x := 2 * (17 + 4); y := x \tag{5.4}$$

Die Klammerung ist notwendig, da wir keine Operatorprioritäten festgelegt haben. Alles in allem liegt der Fokus generell weniger in diesen Details als im Vergleich der Kalküle.

SOS-Formalisierung des Beispiels

Die SOS-Regeln lassen sich direkt über den Produktionsregeln definieren, wenn man diese als Terme betrachtet (s.u.). Für L-Ausdrücke sehen die Regeln gemäß Definition 10 wie folgt aus (Auszug):

$$\text{Add1:} \quad \frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma\rangle}{\langle e_0 + e_1, \sigma\rangle \to \langle e_0' + e_1, \sigma\rangle} \tag{5.5}$$

$$\text{Add2:} \quad \frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma\rangle}{\langle m + e_1, \sigma\rangle \to \langle m + e_1, \sigma\rangle} \tag{5.6}$$

$$\text{Add3:} \quad \langle m + m', \sigma\rangle \to \langle n, \sigma\rangle \text{ mit } n = m + m' \tag{5.7}$$

$$\text{Var:} \quad \langle v, \sigma\rangle \to \langle \sigma(v), \sigma\rangle \tag{5.8}$$

$$\text{Mult1:} \quad \frac{\langle e_0, \sigma\rangle \to \langle e_0', \sigma\rangle}{\langle e_0 * e_1, \sigma\rangle \to \langle e_0' * e_1, \sigma\rangle} \tag{5.9}$$

$$\text{Mult2:} \quad \frac{\langle e_1, \sigma\rangle \to \langle e_1', \sigma\rangle}{\langle m * e_1, \sigma\rangle \to \langle m * e_1, \sigma\rangle} \tag{5.10}$$

$$\text{Mult3:} \quad \langle m * m', \sigma\rangle \to \langle n, \sigma\rangle \text{ mit } n = m * m' \tag{5.11}$$

etc.

wobei $m, n \in \mathbb{N}$ für Metavariablen stehen. Die Regel 'Var' spezifiziert den Zugriff auf den Wert einer Variablen. Die Anweisungen bestehen aus Zuweisung, Komposition

und Kontrollfluss mittels **if** und **while** wie folgt:

$$\text{Nil:} \quad \langle \varepsilon, \sigma \rangle \to \sigma \tag{5.12}$$

$$\text{Assign:} \quad \frac{\langle e, \sigma \rangle \to^* \langle m, \sigma \rangle}{\langle v := e, \sigma \rangle \to \langle \sigma[m/v] \rangle} \tag{5.13}$$

$$\text{Comp1:} \quad \frac{\langle c_0, \sigma \rangle \to \langle c_0', \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to \langle c_0'; c_1, \sigma' \rangle} \tag{5.14}$$

$$\text{Comp2:} \quad \frac{\langle c_0, \sigma \rangle \to \langle \sigma' \rangle}{\langle c_0; c_1, \sigma \rangle \to \langle c_1, \sigma' \rangle} \tag{5.15}$$

$$\text{If1:} \quad \frac{\langle e, \sigma \rangle \to^* \langle 0, \sigma \rangle}{\langle \textbf{if } e \textbf{ then } c \textbf{ else } c', \sigma \rangle \to \langle c, \sigma \rangle} \tag{5.16}$$

$$\text{If2:} \quad \frac{\langle e, \sigma \rangle \to^* \langle m, \sigma \rangle}{\langle \textbf{if } e \textbf{ then } c \textbf{ else } c', \sigma \rangle \to \langle c', \sigma \rangle} \text{ mit } m \neq 0 \tag{5.17}$$

$$\text{while1:} \quad \frac{\langle e, \sigma \rangle \to^* \langle 0, \sigma \rangle}{\langle \textbf{while } e \textbf{ do } c, \sigma \rangle \to \langle c;\ \textbf{while } e \textbf{ do } c, \sigma \rangle} \tag{5.18}$$

$$\text{while2:} \quad \frac{\langle e, \sigma \rangle \to^* \langle m, \sigma \rangle}{\langle \textbf{while } e \textbf{ do } c, \sigma \rangle \to \sigma} \text{ mit } m \neq 0 \tag{5.19}$$

Genau genommen müssten die SOS-Regeln nach unserer Definition 10 auf einer Signatur Σ basieren, die die Produktionsregeln und Kommandos z.B. wie folgt abbildet:

$$\begin{aligned} \Omega =_{def} \quad & + : \textsc{Expr} \times \textsc{Expr} \to \textsc{Expr} \\ & - : \textsc{Expr} \times \textsc{Expr} \to \textsc{Expr} \\ & * : \textsc{Expr} \times \textsc{Expr} \to \textsc{Expr} \\ & v : \textsc{Var} \to \textsc{Expr} \\ & lit : \textsc{Value} \to \textsc{Expr} \\ & := : \textsc{Var} \times \textsc{Expr} \to \textsc{Command} \\ & if : \textsc{Expr} \times \textsc{Command} \times \textsc{Command} \to \textsc{Command} \\ & while : \textsc{Expr} \times \textsc{Command} \to \textsc{Command} \\ & L : \mathcal{P}(\textsc{Command}) \to \textsc{Value} \end{aligned} \tag{5.20}$$

Die Transitionsregeln würden dann über Σ-Termen beschrieben werden. Um das Beispiel möglichst nahe am Original zu halten, wird auf diese Umschreibung verzichtet. Die Regeln angewandt auf das Beispielprogramm 5.4 ergeben demnach die Ableitungssequenz:

$$\langle x := 2 * (17 + 4); y := x, \sigma \rangle \xrightarrow{Assign} \langle y := x, \sigma[42/x] \rangle \tag{5.21}$$

$$\xrightarrow{Assign} \langle \varepsilon, \sigma[42/y] \rangle \tag{5.22}$$

$$\xrightarrow{Nil} 42 \tag{5.23}$$

An der Ableitungssequenz zeigt sich, wie Bedeutend die Errungenschaft von Plotkin ist, dass durch die zweidimensionale Interpretation nur Konklusionen von Regeln und nicht mehr vollständige, lineare Sequenzen für die Herleitung notiert werden (vgl. Abschn. 2.5). Genau genommen haben wir durch die Wahl der Regeln *bewusst* entschieden, dass Ausdrücke nicht in der Ableitungssequenz vorkommen. Blickt man auf die Konfigurationsänderungen σ, so stellt sich die reduzierte Zustandsfolge ein:

$$\underbrace{\sigma[]}_{z_0} \to \underbrace{\sigma[x = 42]}_{z_1} \to \underbrace{\sigma[x = 42, y = 42]}_{z_2} \tag{5.24}$$

So gesehen werden L-Ausdrücke wie in einer funktionalen Sprache ohne Konfigurations- und damit Zustandsänderung interpretiert. Daraus folgt, dass ein Programmablauf

als Ableitungssequenz nicht in allen Fällen einem Trace entsprechen muss, sondern allgemein nur die Anwendung von Transitionsregeln der Form $\langle t, \sigma\rangle \rightarrow \langle t', \sigma'\rangle$ mit $\sigma \neq \sigma'$ zu neuen Zuständen führt. Anders formuliert hängt die Zustandsfolge direkt mit den definierten Transitionsregeln zusammen. Auf diesen Aspekt werden wir im Vergleich der Formalisierungen zurückkommen.

ASM-Formalisierung des Beispiels

Die abstrakte Syntax der Produktionsregeln 5.2 lassen sich nicht direkt zur ASM-Definition verwenden (vgl. Abschnitt 5.1). Die angesprochene Σ-Signatur aus 5.21 könnte zumindest in Teilen verwendet werden, wenn auch die Herangehensweise einer ASM-Formalisierung eher aus Sicht des Zustandes stattfindet, da dieser maßgeblich die Update-Regeln prägt. Ein *L*-Programm ließe sich durch die folgenden Signaturdefinitionen kompakt beschreiben:

domain VARIABLE $=_{def} \{x | x \in \{a...z\}^+\}$
domain VALUE $=_{def} \mathbb{N}$
domain COMMAND $=_{def}$ { Knoten der Produktionsregel c}
domain CMDKIND $=_{def} \{assign, if, while\}$
cmd: $\rightarrow$ COMMAND
next: COMMAND $\rightarrow$ COMMAND
kind: COMMAND $\rightarrow$ CMDKIND
cond: COMMAND $\rightarrow$ EXPR
var: COMMAND $\rightarrow$ VARIABLE
eval: EXPR $\rightarrow$ VALUE
config: VARIABLE $\rightarrow$ VALUE
then: COMMAND $\rightarrow$ COMMAND
else: COMMAND $\rightarrow$ COMMAND
first: COMMAND $\rightarrow$ COMMAND

Die abstrakte Syntax der *L*-Programme besteht in diesem Fall aus Sequenzen von COMMAND-Elementen, bei denen *next*: COMMAND $\rightarrow$ COMMAND das Abarbeiten in sequentieller Reihenfolge sicherstellt.[19] Als Programmzeiger dient die Operation *cmd*, welche durch Updates je nach Voranschreiten eines Programms aktualisiert wird. Weitere Navigation durch die Kommandostruktur bieten die Operationen *cond*, die den Bedingungsausdruck für if/while-Anweisungen zurückliefert sowie *then*, *else* und *first*, die jeweils die erste Anweisung innerhalb der if-, else- und while-„Blöcken“ returnieren. Für Ausdrücke wird eine verkürzte Definition mittels *eval* angenommen und nicht weiter definiert. Darauf basierend kann die Hauptregel wie folgt spezifiziert werden:

[19] Für die while-Schleife nehmen wir an, dass *next* nach letzter Anweisungen innerhalb der Schleife erneut das while-Kommando selbst liefert.

```
Rule main:
if kind(cmd) = assign then
    config(var(cmd)) := eval(expr(cmd))
    cmd := next(cmd)
else if kind(cmd) = if then
    if eval(expr(cmd)) ≠ 0 then
        cmd := then(cmd)
    else
        cmd := else(cmd)
    endif
else if kind(cmd) = while then
    if eval(expr(cmd)) ≠ 0 then
        cmd := first(cmd)
    else
        cmd := next(cmd)
    endif
endif
```

Wie zu sehen wird anhand des Ausdruckstyps CmdKind ein Zustandsautomat als Fallunterscheidung strukturiert. Mit Blick auf die Transitionen modellieren Updates hier einzig die iterative Abarbeitung, die Berechnung von Ausdrücken wird im Kontext einer Zuweisung angestoßen und unmittelbar einer Variablen zugewiesen. Daher sind die Zustandsänderungen für das Beispielprogramm dieselben wie die in 5.24 gezeigten und entsprechen der „zeilenweise“ Abarbeitung von L-Programmen. Natürlich war das eine bewusste Entscheidung, um die Äquivalenzen beider Maschinen zu verdeutlichen. Man hätte in beiden Fällen Ausdrücke ebenso mittels Zwischenzuständen abarbeiten können, z.B. unter Zuhilfenahme eines Stapels für Teilergebnisse der Ausdrucksberechnung. Dies könnte analog zur Aufteilung der Regeln mittels CmdKind modelliert werden.

Allgemein kann diese Art des Vorgehens der ASM-Formalisierung als Entwurfsmuster bei verschiedenen Autoren beobachtet werden (z.B. [83][24][27]). Siehe auch die Diskussion zur Übersetzung in Abschnitt 5.2.3.

MActions-Formalisierung des Beispiels

Das Metamodell für die abstrakte Syntax der Beispielsprache spiegelt die Unterteilung der Produktionsregeln e für Ausdrücke und c für Anweisungen als separate Vererbungshierarchien wider (s. Abb. 5.2). Zusätzlich sind die Zustandsgrößen als Variablen im Laufzeitmodell modelliert. Ein L-Programm wird zur Laufzeit als Instanz von `Process` repräsentiert, wobei der Instruktionszeiger als Referenz `pc` auf den aktuellen `Command` zeigt. Dieses Vorgehen ist im Prinzip analog zu der ASM-Modellierung von *cmd*.

Das Verhalten zeigt Abbildung 5.2. Die Hauptschleife der sequentiellen Abarbeitung von Anweisungen ist in `exec` spezifiziert, die wir für ASM nur informell beschrieben haben. Unsere ASM-Definition der Sprache L ist also so nicht ausführbar, ließe sich aber über eine detailliertere Modellierung korrigieren.

Die Auswertung von Ausdrücken definieren wir mit der in Kapitel 3.4 eingeführten Kurznotation für Operationen, die aus einem einzelnen OCL-Queryausdruck bestehen und die sich in fast identischer Weise auch für ASM (über der Signatur 5.21) beschreiben lassen:

```
context PlusExpr:
def eval(Expr e1, Expr e2) : Integer = eval(e1) + eval(e2)
  redefines Expr#eval(Expr, Expr) : Integer
```

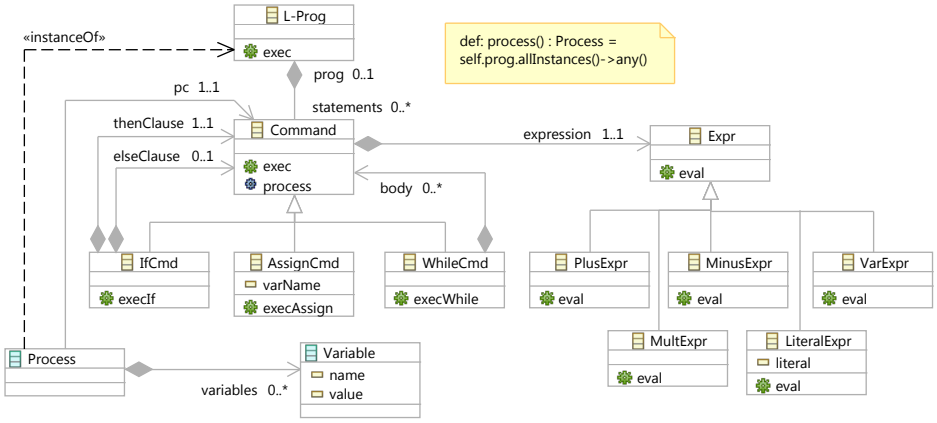


Abbildung 5.2: Metamodell der Sprache L

context MinusExpr:
def eval(Expr e1, Expr e2) : **Integer** = eval(e1) − eval(e2)
  redefines Expr#eval(Expr, Expr) : **Integer**

context MultExpr:
def eval(Expr e1, Expr e2) : **Integer** = eval(e1) ∗ eval(e2)
  redefines Expr#eval(Expr, Expr) : **Integer**

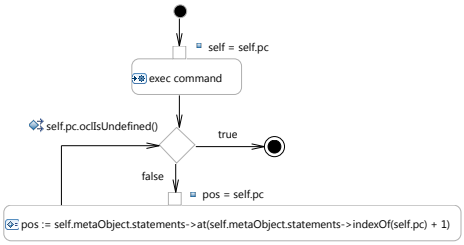
Die Aktionsmodellierung fokussiert im ersten Schritt auf das, was bei der Ausführung eines Sprachkonstrukts passiert, wobei jedwede Zustandsbetrachtung außen vor bleibt. Erst im zweiten Schritt kann man sich die Frage stellen, ob die durch modellierte Aktionen ergebenen (Mikro-)Zustände das gewünschte Verhalten mit Blick auf den Zustand nachstellen, oder ob es einer Anpassung bedarf. Im Beispiel ergeben sich durch die Aktionen zunächst die folgenden Mikrozustandsübergänge, wobei die Laufzeitumgebung (Thread, Stack, Parameter etc.) nicht berücksichtigt ist:

$$\underbrace{\sigma[]}_{\mu z_0} \xrightarrow{\text{create process}} \underbrace{\sigma[p]}_{\mu z_1} \xrightarrow{\text{create var}} \underbrace{\sigma[p, < v >= undef]}_{\mu z_2} \xrightarrow{\text{name var}} \underbrace{\sigma[p, x = undef]}_{\mu z_3}$$
$$\xrightarrow{\text{associate var}} \underbrace{\sigma[p.x = undef]}_{\mu z_4} \xrightarrow{\text{assign}} \underbrace{\sigma[p.x = 42]}_{\mu z_5} \rightarrow \ldots \quad (5.25)$$

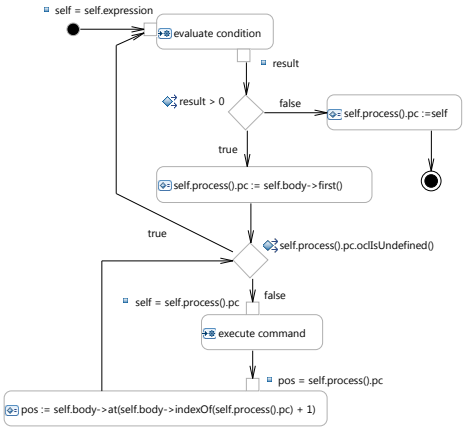
Um die Zustandsfolge aus 5.24 herzustellen, bedarf es einer Annotation von State Generating Transitions (SGT, vgl. Abschn. 4.1.5) an der Hauptschleife in `exec`. Im Vergleich zu SOS und ASM zeigt sich deutlich, wie diese Ebene der Modellierung bei MActions separat ist und die Strukturierung der Metamodelle und Aktionen nicht (oder nur in seltenen Fällen) beeinflusst. Der Modellierer erhält somit die Freiheit, zunächst die operationale Interpretation einer Sprache zu definieren und anschließend für Analysen die tatsächliche Zustandssicht zu definieren. Eine Bewertung von dynamischen Eigenschaften wie „Programm terminiert“, „Berechnung braucht n Rechenschritte“ etc. findet also nachträglich statt und ist entkoppelt von der eigentlich Semantikdefinition durch Aktionen.

5.2.1 Zustandsebenen und Zustandsübergänge

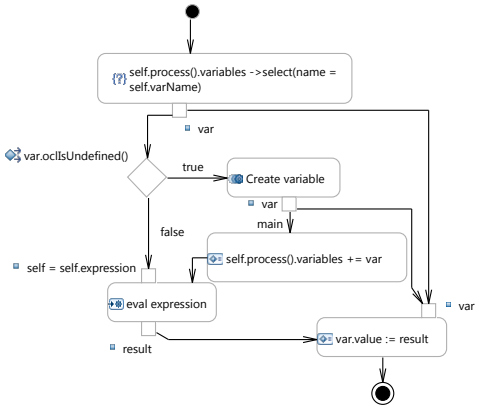
Abgesehen von unterschiedlichen Strukturierungsmerkmalen drücken Transitionsregeln in SOS, ASM und MActions oberflächlich gesehen dasselbe aus: die Transition



(a) Hauptiteration



(b) While-Schleife



(c) Zuweisung

Abbildung 5.3: Auszug der Anweisungsverarbeitung

von einer Konfiguration zur nächsten. Die bestehenden Unterschiede der Ausdrucksweise der Transitionsregeln wurden im vorangehenden Abschnitt diskutiert. Wie Tabelle 5.1.1 zeigt, realisieren die Kalküle allerdings unterschiedliche Maschinenparadigma, was sich zum Teil darin äußert, dass Laufzeitstrukturen implizit Teil des Zustands einer Sprache sind. Dieser Zusammenhang, zwischen den Zuständen der Sprachebene und der Modellebene, soll im Folgenden analysiert werden.

Im Fall von ASM bestimmt ein Satz von Update-Regeln, ob und wie vom aktuellen Zustand in einen Folgezustand übergegangen wird. Während SOS/ASM also den Zustand eines Modells komplett in Termen/Algebren halten, bedarf es bei der MActions-Ausführung nicht nur des Laufzeitmodells der jeweiligen abstrakten Syntax, um den nächsten Zustand zu bestimmen, sondern der kompletten M3-Laufzeitstrukturen. Zum Beispiel bestimmt beim Erreichen eines *Final Node* eines Aktivitätsflusses der Stapel des M3-Laufzeitmodells, welche Aktion als nächstes ausgeführt wird („Rücksprungadresse“). Somit kann ein Satz von Regeln nicht einfach auf einen einmal festgehaltenen Zustand *ohne Kontext* angewendet werden, da bei einer Stapelmaschine immer ein Teil der Ausführungshistorie nötig ist, um den nächsten Schritt zu bestimmen. Als Konsequenz ist demnach der Zustand des M3-Stapel implizit Teil des Zustands des Laufzeitmodells einer Sprache auf M2. Auf diesen Punkt kommen wir noch zurück, jedoch sollte vorher eine Analyse der Zustandsebenen erfolgen. Es sei angemerkt, dass neben ASM auch andere Ansätze

„zustandslose“ Regeln verfolgen, wie z.B. Modelltransformationen mittels Graphersetzung basierend auf Mustererkennung im Objektgraphen.

Greifen wir die Unterteilung der Zustände in Mikrozustände, interne und externe Zustände aus Kapitel 4.1.2 auf, so ergeben sich die Mikrozustände eines Laufzeitmodells auf M2 genau durch jene Übergänge und Veränderungen der M3-Laufzeitumgebung. Das heißt sobald eine zustandsverändernde Aktion ausgeführt wurde, wird ein neuer Mikrozustand erreicht. Wie in Kapitel 4.4.1 gezeigt, kann durch State Generating Transitions die gewünschte „Zustandsgranularität“ eingestellt werden.

Blickt man in Richtung der M1-Ebene, so ergeben sich weitere, vom Anwendungskontext der M1-Modelle abhängige, relevante Zustände. Im Beispiel des Steamboiler war dies z.B. der Zustand *Regulating*, der besondere Aufmerksamkeit bei der Modellierung bedurfte, weil nach fünf Sekunden und Nichteintreten des gewünschten Wasserpegels die Steuerung anzuhalten war (vgl. 4.4.1). Daraus ergibt sich, dass diese ausgezeichneten Zustände in M1-Modellen keine direkte Berücksichtigung in der Sprachdefinition finden können. Andererseits muss es dem Anwender einer Sprache aber möglich sein, über die Zustände der M2-Ebene (also dem Laufzeitmodell) solche *herzuleiten*. Nichts anderes geschah beim Beispiel des Steamboilers mit der Navigation durch das Laufzeitmodell, z.B. mittels `cp.activeState = 'Regulating'`. Aus dieser Analyse heraus folgt eine Zuordnung der (zunächst intuitiv festgestellten) Zustandsübergänge zu den Metaebenen des MOF-Frameworks:

M1: Ausgezeichnete Zustände im Anwendungsmodell
M2: Zustände durch konkrete MActions (ggf. definiert mit State Generating Transitions)
M3: Mikrozustände durch Laufzeitmodell der `MActions`

Diese Zuordnung zu den Metaebenen M1 bis M3 lässt sich verallgemeinern, so dass man generell für eine Ebene M_n sagen kann, dass eventuelle Mikrozustände immer durch Änderungen der Ebene M_{n+1} zustande kommen. Liefe zum Beispiel die in Kapitel 3 gegebene Aktionsdefinition in einem Interpreter ab (d.h. die M4-Ebene würde bei der Simulation mit ausgeführt), so unterteilen sich Mikrozustände der Ebene M3 automatisch in weitere Mikrozustände der Ebene M4. Ein Mittel diese zu verhindern wären wie gezeigt State Generating Transitions, so dass ein Zustandsraum ohne Zwischenzustände realisierbar ist. Daraus folgt, dass diese Zuordnung dem intuitiven Verständnis der Ausführung eines realen Programms durch eine Maschine entspricht. Zum Beispiel durchläuft ein Simulator, der ein Simulationsmodell ausführt, nicht nur die Programmzustände der Implementierung des Simulators, sondern ferner Zwischenzustände, die durch die Rechnerarchitektur vorgegeben sind (Registertransfers, Speicherzugriffe), diese wiederum sind bestimmt durch physikalische Prozesse der Elektrotechnik u.s.w. Somit bietet das εMOF-Framework mit einer operationalen Sprache wie den MActions ein praktisches Denkmodell, in den sich diese Prozesse einordnen und charakterisieren lassen.

Zurückkommend auf den anfangs begonnen Vergleich mit ASM stellt sich die Frage, ob diese „Mikrozustände“ nur ein Phänomen der MActions (bzw. MOF) sind, oder aber genereller Natur. Dazu kann man feststellen, dass zustandslose Ansätze auf den ersten Blick eine striktere Trennung der Zustandsebenen vornehmen. Im ASM-Kalkül existieren keine „Mikrozustände“, sondern lediglich abstrakte Zustände in linearer Folge als Resultat von Updates, die ohne Wissen über vorherige Schritte angewandt werden. So gesehen ist die Entwicklung der Algebren unabhängig von einer „ASM-Metaebene“, auf der Zwischenresultate erzeugt werden. Anders formuliert, bedeutet dies, dass die Zustandsschritte der Metaebene (ASM) mit denen der Objektebene (beschriebene Sprache) *koinzidieren*.

Umgekehrt bedeutet dies, dass für eine Definition einer beliebigen Sprache (!) jeder Zustandsübergang mittels *eines einzigen* Updates beschrieben werden können muss; natürlich mit einer beliebigen (endlichen) Menge von Regeln. So müsste z.B. eine Definition von ASM *mit* ASM sämtliche Update-Regeln wiederum mit einer einzigen Menge von Updates beschreiben, damit die Zustandsübergänge der „Metasprache“ ASM (gedacht auf M2) zur Definition der „Objektsprache“ (gedacht auf M1) koinzidieren. Da eine Transition im Prinzip jeden beliebigen Algorithmus ausdrücken kann, müsste man mit einer Regelmenge also jede berechenbare Funktion ausdrücken können, ohne Zwischenschritte zu nutzen. Ein Erreichen dieser Übereinstimmung wäre (für die meisten Sprachen) vermutlich schwierig, wenn nicht gar unmöglich.[20]

Daraus gefolgert kann man festhalten, dass Mikrozustände relevant werden, sobald in der Metasprache die Definition der Ausführungssemantik im Hinblick auf Zustandsübergänge nicht eins zu eins gelingt. Bei der Definition der MActions wurden SGTs nicht angegeben, da man ohne dynamische Analyse oder anderen Formen der Auswertung losgelöst von den Zuständen agieren kann. Hier könnten unterschiedliche Interpretationen ansetzen, um die Mikrozustände für M2 von vornherein einzuschränken oder zu eliminieren. In jedem Fall ermöglichen SGTs eine frei wählbare Konfigurationstransition, die die Zusammenfassung von beliebigen Aktionen, und damit beliebigen Algorithmen während einer Transition, erlaubt.

Beobachtbare Zustände, Modellzeit

Plotkin und Gurevich unterscheiden beide bereits in ihrer Herangehensweise *internes* und *externes* Verhalten bei der operationalen Semantikbeschreibung. Externes Verhalten wird dabei solange als irrelevant betrachtet, bis ein Eingriff in den Ablauf eintritt (vgl. [21],[97]). Diese im Hinblick auf Zustände zunächst orthogonale Betrachtung wurde in Kapitel 3 und 4 aufgenommen und in Zusammenhang mit der Zustandstransition gesetzt. Externes Verhalten, das durch InputActions in einen Ablauf einfließt, wird generell als nicht deterministisch erachtet. Es kann somit keine Aussage über eingehende Objekte getroffen werden, außer deren Typ (durch getypte Eingabepins). Im Vergleich zu ASM verhalten sie sich also in etwa wie **shared**- bzw. **monitored**-Funktionen, die allerdings in MActions keine direkte Kopplung mit dem nächsten Zustand[21] erfahren. Nachfolgende Aktionen können Eingabedaten zunächst auswerten und durch SGTs entscheiden, wann und wie diese Eingabedaten den aktuellen Zustand beeinflussen. Dadurch verlagern sich implizite Annahmen über Eingangsdaten in explizite Auswertungen in der Definition einer operationalen Semantik.

Für Prozesskalküle wie CCS werden die Ein- und Ausgaben in der Regel wichtig, um *Prozesskommunikation* auszuwerten (z.B. [88]). Verteilte ASM fallen in die gleiche Kategorie von Sprachen, bei denen ein Zustandsübergang durch Nachrichtenaustausch ausgelöst werden kann. MActions folgen hier einem anderen Muster, da Ein- und Ausgaben *immer* nur an die Umgebung erfolgen. Die Gesetzmäßigkeiten dort entziehen sich dem Ablauf innerhalb des LZR und ASR und können nicht

[20] Dem Autor sind keinerlei Arbeiten dahingehend bekannt, ASM metazirkulär zu definieren, da ASM und vergleichbare Kalküle eigentlich immer ausschließlich durch mathematische Definitionen beschrieben werden. Allerdings kann man sich für Funktionskalküle durchaus vorstellen, dass sie diese Eigenschaft mit sich bringen. Für ASM wurde z.B. Rekursion in [141] über „Agenten-Stapel“ definiert, bei denen jeder Subagent jeweils eine neue Inkarnation ausführt. Weitere Arbeiten sind die von Börger in [111], wo er sog. *micro steps* von Agenten einführt, die mit Mikrozustandsübergängen verglichen werden können.

[21] Allerdings ist ein Mikrozustand nach eingehen im LZR natürlich vorhanden

innerhalb der operationalen Semantik reflektiert werden. So kann eine Ausgabe an die Umgebung z.B. zum Anhalten der gesamten Ausführung im Simulator führen, auch wenn nach der Ausgabeaktion weitere Aktionen folgen würden. Andere Formen der Kommunikation zwischen Prozessen (Threads) sind nicht in der Sprachdefinition von MActions enthalten. Sollte Kommunikation erwünscht sein, so muss sie explizit *modelliert* werden und z.B. mit verschiedenen Threads oder mittels Rundlaufverfahren realisiert werden.

Gleiches gilt für die Simulationszeit, logische Uhren oder Synchronisation von Uhren: die Ausführungsumgebung als Universelle Transformationsmaschine schränkt die Definition nicht weiter ein, da sowohl Kausal nacheinander auftretende Ereignisse (mittels Aktionen) sowie echte Parallelität durch die Axiome abgesichert und modellierbar sind. Sind Uhren/Zeitverläufe also als Funktion beschreibbar, können sie mit MActions modelliert werden. Ein Beispiel von lokalen Uhren (Eigenzeit) ist in Kapitel 4.4.1 beschrieben.

5.2.2 Parallelität

Parallelitätseigenschaften werden für die Spezifikation und Entwicklung von Sprachen immer wichtiger, nicht zuletzt aufgrund von aktuellen Hardwareentwicklungen hin zu Mehrprozessorkernen und somit für die dafür zu modellierende Anwendungssoftware. Aus diesem Grund sollte die Modellierung von Nebenläufigkeitseigenschaften und *true concurrency* in der operationalen Semantik Unterstützung finden.

Während Milner ursprünglich für CCS die Gleichzeitigkeit von Ereignissen unter der Annahme ausschloss, dass ein außenstehender Betrachter nur ein Ereignis zur Zeit wahrnehmen kann (vgl. [88]), führen die meisten Kalküle zu diesem Zweck eine Art „Parallelitätsoperator“ ein. Beispiele dafür sind CSP von Hoare [89] (mit $P \parallel Q$) oder Plotkins *parallel OR*-Operator.Der Blickwinkel liegt durch die Art der Schlussregeln als Transitionssysteme *global* auf den „Aktionen“ (also Bezeichnern, s. 2.5), um generelle Aussagen über parallel ausgeführte Programme zu formulieren (vgl. [96]). Insbesondere um die Fragestellung beantworten zu können, ob sich zwei Programme gleich verhalten, wird Prozessverhalten durch Reduktion auf Synchronisationsbäume soweit abstrahiert, dass sich die Frage durch Termkongruenz beantworten lässt. In der Praxis stehen allerdings oftmals Fragen nach der *exakten Konfiguration* im Vordergrund und *wie* Programmkonstrukte Zustandsgrößen parallel verändern und ggf. interferieren können (*race conditions*). Dazu bedarf es unserer Ansicht nach einer eher *lokalen* Modellierung, die jeweils aus Sicht der Prozesse spezifiziert, welche Schritte z.B. beim Zugriff auf exklusive Ressourcen durchlaufen werden, wie sich Warteschlangen von Prozessen füllen und das Scheduling geschieht etc. Geht man einen Schritt weiter und betrachtet man Fairness-Eigenschaften, so sind einige Kalküle durch fehlende Ausdrucksmöglichkeiten limitiert und es bedarf Erweiterungen, um diese Problematik zu adressieren (vgl. [142] für CCS und CSP).

Aus dieser Motivation heraus wurde der Aktionsraum und die AtomicGroup-Semantik in die MActions aufgenommen, die durch Konsistenz- und Kohärenzkriterium eine Vielzahl an möglichen Parallelitäts- und Nebenläufigkeitscharakteristiken modellierbar machen. Neben der für C# gezeigten Spezifikation des Threading in der Common Language Runtime von .NET (s. Anhang A) und der als *round robin* modellierten Timed Automata (s. 4.4.1) lassen sich prinzipiell auch Scheduling-Algorithmen formulieren, die unterschiedliche Fairness- oder Priorisierungseigenschaften besitzen. Da wir nur an endlichen Ausführungsläufen interessiert sind, hat Fairness in unserem Kontext allerdings eine eingeschränkte Aussagekraft, da es z.B. bei einem Ausführungslauf von einer fairen Schedulingstrategie aufgrund

des endlichen Traces zu Ungleichgewichten kommen kann (die sich allerdings durch verlängerte Läufe beliebig verkleinern lassen sollten).

5.2.3 Denotationelle Semantik durch Übersetzung in Operationale Kalküle

Aus der beispielhaften Formalisierung im Kapitel 5.2 wird im Besonderen der operationale Charakter der Semantikdefinitionen deutlich. Wohlgemerkt entspricht diese Vorgehensweise im Allgemeinen der Definition eines *Interpreters* und nicht einer *Übersetzung* in ein Kalkül im Sinne eines Kompilierungsvorganges (s. Abschn. 5.1). Natürlich könnte ein Programm in ein operationales Kalkül übersetzt werden, anstatt es schrittweise zu interpretieren, zum Beispiel das L-Programm 5.4 sähe in ASM in etwa wie folgt aus:

```
Rule main:
do seq
    x := 2 * (17 + 4)
    y := x
enddo
```

Dadurch würde die gängige Denotation mittels Funktionen ersetzt durch ein operationales Kalkül, in Kurzform also: $[\![AS]\!] = \Sigma, \mathbb{A}; \langle \Sigma, \mathbb{A} \rangle_{\mathbf{EA}}$. Dieser Ansatz hat den Vorteil, dass Programme in sehr kompakter Form in der Zieldomäne vorliegen. Allerdings verliert man im Allgemeinen aufgrund komplexeren Regeln der Abbildung den Bezug einzelner Konstrukte der (abstrakten) Syntax zu ihrer individuellen (Teil-)Semantik.

5.3 Verwandte Arbeiten

Verschiedene Forschungsgruppen haben Vorschläge und Ansätze zur Problematik der operationalen Semantik in MOF erarbeitet, welche in diesem Abschnitt mit dem Ansatz der MActions verglichen werden sollen. Viele der Arbeiten sind im Kontext der Evolution von UML1.x zu UML2.x entstanden, die Einherging mit der Standardisierung von MOF 2.x bei der OMG als zugrundeliegendes Metamodellierungsframework [143][144]. Die konzeptionelle Herangehensweise führt in der Regel zu einer Einteilung in Aktionssprachen, (Modell-)Transformationssprachen und in Frameworks eingebettete Sprachen, die im Folgenden in Relation zu dieser Arbeit gesetzt und diskutiert werden.

5.3.1 Aktionssprachen für MOF

Eine ganze Reihe von Ansätzen verfolgt mittels Aktionssemantiken relativ ähnliche Konzeptionen zu operationaler Semantik für MOF. Kern ist die Verknüpfung von modellverändernden Aktionen mit Operationen in Metamodellklassen. Hierzu zählen die Arbeiten von Clark et al. zu XMF [39] und Fleurey et al. zu KerMeta [38], die hier stellvertretend diskutiert werden sollen. Generell ist diesen Sprachen eine Definition von elementaren Aktionen gemein, die Metamodellinstanzen verändern können und somit eine objektorientierte, ausführbare Spezifikationssprache bieten. So erlaubt z.B. KerMeta das Generieren von Javacode aus den Aktionsdefinitionen, wodurch annotierte Metamodelle in einem Schritt zu ausführbaren EMF-Implementierungen werden, die als Basis der Entwicklung von Simulatoren dienen können. Größtenteils wird für Ausdrücke entweder ein Derivat von OCL verwendet (z.B. XOCL bei

XMF) oder proprietäre Syntaxformate (z.B. MetaEdit+, vgl. [145]). Diese Ansätze sind generell Implementierungen textueller Skriptsprachen, die MActions-ähnliche (elementare) Aktionen bereitstellen, allerdings ohne eine formale Definition anzugeben. Ausführungsaspekte von MOF-Modellen sind somit faktisch durch die Zieldomäne festgelegt, in die generiert oder in der implementiert wird. Deshalb bleiben weiterführende Aspekte wie Zustandsübergänge, Parallelität oder auf den Beschreibungen aufbauende Modellanalysen außer Betracht. Dies ist verständlich, da der Anwendungsfokus im Entwicklungskontext liegt.

Weitere Ansätze nutzen reine UML-Aktionen, Untermengen davon oder eine Kombinationen mit Programmiersprachen (z.B. [146][121]). Für letzteres zeigt z.B. Scheidgen eine repräsentative Fallstudie für SDL, bei der ein Großteil der Sprachsemantik in einem SDL-Metamodell in Java implementiert wurde [71].[22] Ein weiteres Beispiel ist die fUML-Spezifikation (*Semantics of a Foundational Subset for Executable UML Models*, [147]), die eine eigens formalisierte Untermenge von Java zur Definition verwendet. Generell bestätigen diese Arbeiten den Bedarf nach pragmatischen Ansätzen zur Definition von ausführbaren Modellen, aus denen Werkzeuge automatisiert abgeleitet werden können.

5.3.2 Modelltransformationen zur operationalen Semantik

Eine zweite Gruppe von Ansätzen greift auf Techniken von Graphersetzungssystemen zurück, um Veränderungen an MOF-Modellen und somit operationales Verhalten zu spezifizieren. Als frühes Beispiel für diese Art der Semantikbeschreibung zählen die Arbeiten von Engels et al. zur *dynamischen Metamodellierung* [29]. Dort finden (normale) Graphersetzungsregeln Anwendung, bei denen der Mustergraph auf „MOF-Arbeitsgraphen“ angewandt wird, um eine Veränderung zu bewirken.

Ein genereller Vorteil dieser Herangehensweise besteht in der mathematischen Fundierung der Graph- und Ersetzungstheorie. Da MOF-Modelle letztlich attributierte Graphen darstellen, kann ein Gros dieser Theorie ohne Umwege direkt angewendet werden. Dadurch erhalten nicht nur Regeln eine formale Transitionssemantik, sondern andere Aspekte, wie die Anwendung von parallelen Ersetzungsregeln anhand der *Double-Pushout*-Konstruktion, ist möglich (vgl. [134]). Weiterentwicklungen dieser Idee unter Einsatz des QVT-Transformationsstandards kann in verschiedene Richtungen beobachtet werden (vgl. [148][149]), weil Modelltransformationssprachen vielfach wiederum auf klassischer Graphersetzung oder Triple-Graph-Grammatiken aufsetzen.

Im Vergleich zu einem festen Satz von Aktionen kann eine Graphersetzungsregel beliebige Manipulationen vornehmen. Mehrere Objekte können gleichzeitig erzeugt, verändert oder gelöscht werden, inkl. ihrer Relationen/Referenzen. Die Schwierigkeiten ergeben sich generell bei der Steuerung der Regelanwendung, d.h. welche Regel wann und wie oft ausgeführt wird. Zu diesem Zweck kommen meist in Kombination andere Programmierparadigmen zum Einsatz, die Kontrollstrukturen erlauben (vgl. [150]). Dadurch verändert sich der Charakter der Sprache u.U. stark in Richtung einer Aktionssemantik, auch wenn die Transitionsregeln durch Graphersetzungen erhalten bleiben.[23] Im Hinblick auf Zustandsübergänge finden sich große

[22] Die dort enthaltene *MOF Action Semantics* war gemeinsamer Forschungsgegenstand mit dem Autor für die Entwicklung von testbaren DSLs, vgl. [123]

[23] Ein weiterer untergeordneter Aspekt ist die Laufzeiteffizienz von Graphersetzungssystemen. Da diese auf Musterkennung in Graphen basieren (was an sich ein NP-vollständiges Problem darstellt), können Regelanwendungen teuer und laufzeitineffizient sein. Techniken wie Typisierung von Knoten helfen, um dieses Problem in der Praxis in den Griff zu bekommen.

Gemeinsamkeiten mit den Gesetzmäßigkeiten aus Abschnitt 5.2.1.

5.3.3 Frameworks und UML basierte Ansätze

Über „reine“ Sprachen und Kalküle hinaus gibt es eine Reihe von Werkzeugen und Frameworks, die durch Rückgriff auf existierende Sprachimplementierungen operationale Semantik in Verbindung mit Metamodellen realisieren. So ist z.B. die *ATLAS Model Management Architecture* (AMMA [151]) zu nennen, die die Transformationssprache ATL in ein Metamodellierungsframework einbindet [40]. Noch flexibler ist das von Sadilek et al. konzipierte EProvide zur Einbettung diverser Sprachen durch Adapter in einer eclipse (EMF-)Umgebung, die auf einer 'big step'-Semantik operieren[24] (vgl. [152]). Diesen Frameworks ist gemein, dass die eigentlichen Sprachen unverändert bleiben und Transitionsregeln lediglich MOF-Modelle „als Daten“ verarbeiten. Das heißt das MOF-Meta-Metamodell muss nicht erweitert werden, sondern die Beschreibung der operationalen Semantik koexistiert parallel neben der Metadatenarchitektur. Der Vergleich zu MActions fällt damit schwer, da sich die Semantikdefinitionen der MOF-Architektur entziehen und eine separate semantische Domäne bilden, was am ehesten der Abbildung aus Abschnitt 5.2.3 entspricht, also dem Übersetzungsprozess in eine andere Sprache.

Darüber hinaus können im Prinzip Formalisierungen von UML zur Definition von dynamischem Verhalten dienen, weil MOF faktisch eine Untermenge der Sprache bildet.[25] Neben der *Executable UML* (xUML), welche im Prinzip zurückgeht auf Methoden von Shlaer, Mellor, Booch und Rumbaugh als wesentliche Vorarbeiten zu UML1.x und damit auf formale Ansätze, existiert seit 2011 der Industriestandard fUML (vgl. [147]). Die darin enthaltene Untermenge von Aktionen und Aktivitäten gleicht den MActions und kann als weiterer Indikator für die Akzeptanz von aktionsbasierter Semantikdefinition für komplexes Sprachverhalten gewertet werden. Allerdings unterscheidet sich die Formalisierung dieser fUML-Aktionen, da deren semantischer Kern bUML („*Base UML*“) nicht meta-zirkulär, sondern durch ein axiomatischen Kalkül (eigentlich denotationell) beschrieben ist. Auf einen detaillierten Vergleich einzelner Aktionen verzichten wir im Rahmen dieser Arbeit, da er den vorausgegangenen Diskussionspunkten keine neuen Erkenntnisse hinzufügt.

5.4 LT-OCL und Dynamische Analysen

Bisherige Ansätze zur dynamischen Analyse basieren generell auf einer Modellierungssprache mit operationaler Semantik, der im Prinzip eine Transitionssemantik innewohnt. Neben den in Kapitel 4.4.1 beispielhaft gezeigten *Timed Automata* kommen fast immer ähnliche Techniken der Zustandsmodellierung zum Einsatz, bei denen dann mittels Temporallogik dynamische Eigenschaften formal über Zustandssequenzen (oder Automaten, Kripke-Strukturen) formuliert werden. Kapitel 4 zeigt, wie dieser Ansatz allein mittels Laufzeitmodell generisch für verschiedene Sprachdefinitionen genutzt werden kann. Das Vorgehen für die Sprache LT-OCL diente als komplexeres Beispiel und es scheint plausibel, dass andere Temporallogiken in ähnlicher Form adaptiert werden könnten, so zum Beispiel Lamports *Temporal Logic of Actions* (TLA [101]). Der Grundgedanke einer Auswertung von Ausdrücken und in Relation setzen verschiedener Zustände kann auch hier durch den *prime*-Operator („'“) beobachtet werden.

[24] nicht zu verwechseln mit der *big step* Semantik von McCarthy, vgl. [86]

[25] Als Vorschlag z.B. in [146], siehe auch die Diskussion zu Metaebenen in [147]

Die Idee, OCL um temporale Operatoren zu erweitern, ist hierbei nicht gänzlich neu. Allerdings konzentriert sich der Schwerpunkt bisheriger Arbeiten ausschließlich auf dynamische Eigenschaften von UML-Modellen (vgl. [153][154]). Neben diesen Vorschlägen ohne formale Semantik wurde mit *Temporal OCL* (TOCL) eine formale Definition von linearen temporalen Operatoren *always*, *sometime*, *always-until*, *sometime-before* und *next* von Ziemann et al. für OCL erarbeitet (vgl. [155]).[26] Der hier vorgestellte Ansatz zu LT-OCL entstand zwar davon unabhängig und entwickelt die Ideen über UML hinaus, verfügt aber nur über zukunftsbezogene Ausdrücke (d.h. kein *previous* oder *alwaysPast*, wenn auch als Erweiterungen denkbar). Zudem betrachten wir einen offene Systemsicht mit Blick auf die Interpretation entlang von Traces sowie eine klare *oclUndefined*-Semantik für die zeitliche Auswertung der Formeln.

In Richtung der automatischen Modellprüfung von zeitlichen OCL-Ausdrücken sollte die Arbeit von Distefano genannt werden [156]. Dabei werden OCL-Definitionen auf die *Object-Based Temporal Logic* (BOTL) als semantische Domäne zurückgeführt (vgl. [157]). BOTL ist eine formale Sprache bestehend aus statischen Ausdrücken (ähnlich algebraischer Strukturen) und CTL. Die Semantik von BOTL ihrerseits wird durch Kripke-Strukturen beschrieben (vgl. Abschnitt 2.7). Auch hier konzentriert sich die Anwendung auf die Sprache UML, für die ein Modellchecking eingeschränkt auf State Machines erreicht werden soll. Darüber hinaus existieren verwandte Arbeit für Echtzeit-Systeme mittels verzweigter Zeitlogik und OCL durch Flake et al. (z.B. [158][132][159]).

[26] Die Autoren nutzen dazu ebenfalls eine Version des Objektmodells von Richters et al., vgl. Kapitel 2.4

KAPITEL 6

Zusammenfassung

Ausgehend vom Metamodellierungsstandard MOF wurde die Aktionssemantik MAcTIONS entwickelt, um für Metamodelle eine Sprache zur Definition von operationaler Semantik zu erhalten. Diese Erweiterung um eine Aktionssemantik ging einher mit der Untersuchung, wie Zustände, Zeit und Parallelitätseigenschaften von Sprachen sich so in der operationalen Semantik ausdrücken lassen, dass anschließend dynamische Modellanalysen möglich werden. Zu diesem Zweck wurde mit LT-OCL exemplarisch eine prädikatenlogische Temporallogik entwickelt, die die metamodellunabhängige Analyse von Sprachen über ein um Definitionen zur operationalen Semantik ergänztes Metamodell erlaubt. Als Anwendung der Konzepte konnten mittels einer prototypischen Implementierung zwei Fallstudien durchgeführt werden (Timed Automata und C#), die die Nutzung und das Potential des gesamten Ansatzes aufzeigen.

In Bezug auf die in Kapitel 1.3 beschriebene Zielsetzung dieser Arbeit ergeben sich zusammengefasst die folgenden Ergebnisse für die einzelnen Teilfragen:

zu 1. *Bereitstellung geeigneter Modellierungstechniken für MOF-Metamodelle zur Definition operationaler Semantiken und Modellsimulation*:
Der Schlüssel zur operationalen Semantik für MOF liegt in der Definition von Laufzeitmodellen und der Einbettung dieser in die Metadatenhierarchie. Mittels ausgezeichneter Metaklassen in einem *Laufzeitmetamodell* lassen sich Zustandsgrößen als Basis für Verhalten modellieren. Dafür konnte über verschiedenen Experimente eine *logische Instanzbeziehung* ausgemacht und über ein neu erarbeitetes, zur Metadatenhierarchie orthogonales, MOF-Instanzmodell ins Framework eingegliedert werden (vgl. Kap. 2.3). Zur Beschreibung des operationalen Verhaltens werden elementare Aktionen durch MAcTIONS als Modelltransformation definiert, die Modellsimulation im MOF-Kontext erlaubt (vgl. Kap. 3).

   (a) *MOF-Erweiterungen für die Definition von operationaler Semantik*:
   Die MACTIONS als MOF-Erweiterung fügen sich als Sprache nahtlos in die bislang semantisch nicht weiter spezifizierten Operationen ein, die neben einer klaren operationalen Semantik auch die Basis zur dynamischen Modellanalyse legen (vgl. Kap. 3.1). Die Transitionssemantik über Modelländerungen, Parallelitätseigenschaften und ferner die Wiederver-

wendung der UML-Aktionssyntax ergeben eine graphische Spezifikationssprache, die für die gezeigten, teilweise komplexen Fallbeispiele erfolgreich angewendet werden konnte (vgl. Kap. 4.4.1, A). Als turingvollständige Kernsemantik lassen sich darauf aufbauend semantisch reichere Aktionen entwickeln (vgl. Kap. 3.3,).

(b) *Anwendung und Vergleich zu existierenden Kalkülen/Methoden*:
Im Gegensatz zu SOS oder ASM stellen MActions die *Modelländerung* über die Definition der *Zustandstransition* (vgl. Kap. 2.5.4). Neben dem generellen Paradigma einer Stapelmaschine unterscheiden sich die Ansätze ferner in den Aspekten (1.) Strukturierungsmöglichkeiten und Aufbau der Regeln, (2.) der Formalisierung der Zustandsgrößen, (3.) Beschreibung und Semantik von parallelen Ausführungen sowie (4.) der Herangehensweise an die Kernsemantik und deren Erweiterungsmöglichkeiten (vgl. Kap. 5.1.1). Prinzipiell können wie gezeigt formale Kalküle der Art SOS oder ASM für Metamodelle genutzt werden, allerdings ergeben sich bei der Zusammenführung der unterschiedlichen Beschreibungsformen Brüche (Metamodelle vs. Algebren), die nicht zuletzt mit der objektorientierten Struktur und der Metadatenhierarchie zusammenhängen (vgl. Kap. 5.1). Eine denotationelle Überführung in andere Kalküle ist natürlich denkbar, bildet aber den „Kontrapunkt“ zur Fragestellung in dieser Arbeit (vgl. Kap. 5.2.3). Wir erwarten ferner, dass die Akzeptanz bei Sprachdesignern durch die Anwendung der weit verbreiteten UML-Syntax höher ausfällt, da die aufgezeigten notwendigen Reformalisierungen wegfallen. Ein vergleichendes Beispiel für die Kalküle SOS, ASM und MActions wurde in Kapitel 5.2 gegeben.

(c) *Einordnung und Bewertung in Bezug auf die Modellsimulation*:
Der Schwerpunkt zur Modellausführung lag in dieser Arbeit auf Software(modell)beschreibungen die sich als zeitdiskrete, (nicht-)deterministische, parallele und ereignisbasierte Modellsimulation charakterisieren lässt (vgl. Kap. 1.1.5, 4.1). Die Aktionsdefinitionen ermöglichen eine direkte Umsetzung in einer prototypischen Implementierung als Interpreter, um Experimente sowie Fallstudien durchzuführen (vgl. Kap. 4.5). Im Fokus der dynamischen Analyse stehen die im Simulationskontext üblichen Aussagen über einen Ausführungslauf (Experiment), die dann u.U. durch mehrfache Wiederholung und Anpassung von Parametern verfeinert und validiert werden. Eine Anwendung der Konzepte auf andere Formen der Simulation oder algorithmische Auswertung von Traces wurde nicht untersucht, auch wenn dieses eine konsequente Weiterführung der Untersuchung wäre (vgl. Diskussion 4.1.1).

zu 2.: *Entwicklung einer metamodellunabhängigen Methode zur dynamischen Analyse von MOF-Modellen*:
Das Laufzeitmodell und dessen Zustandsänderungen wurden als Basis einer dynamischen Analyse identifiziert und über ein metamodellunabhängiges Tracemodell zur Aufzeichnung von Ausführungsläufen beschrieben (vgl. Kap. 4.1.3). Es konnte gezeigt werden, dass Verhalten entkoppelt von den Definitionen einer Aktionssemantik ausgewertet werden kann, wenn lediglich das Laufzeitmodell und dessen Änderungen aufgezeichnet und ausgewertet werden (vgl. Kap. 4.1.1). Ferner wurde die enge Beziehung zur operationalen Semantik in MOF durch MActions (oder anderen möglichen Sprachen)[1] über ein erwei-

[1] die den Axiomen der Klasse von Aktionssemantiken gemäß Abschnitt 3.1.2 folgen

tertes Zustandsschema hergestellt, das sich in die Metamodellierungshierarchie einfügt (vgl. Kap. 5.2.1). Zudem konnte für den Verhaltensvergleich von Modellausführungen die Relation zur Bisimulationstheorie aufgezeigt werden (vgl. Kap. 4.2.1). Damit sind die Grundlagen für dynamische Analysen im Kontext von MOF gelegt.

(a) *Wie können dynamische Eigenschaften über beliebige operationale Semantiken und Metamodellen beschrieben werden?*:
Mit der entwickelten zukunftsbezogenen Temporallogik LT-OCL können dynamische Eigenschaften semantisch über dem Tracemetamodell definiert werden, indem das implizierte (generische) Zustandsschema als Basis der Temporalsemantik Anwendung findet (vgl. Kap. 4.3). So wie MOF als Meta-Metamodell die Struktur von Metamodellen festlegt, definiert das Tracemetamodell die möglichen Pfade, die eine Modellsimulation hinterlässt. Diese zustandsbezogene Sicht auf die Ausführung lässt sich bei der von uns betrachteten Klasse von Aktionssemantiken relativ unabhängig bilden, so dass es generell plausibel erscheint, weitere Sprachen nach dem gleichen Schema zu definieren (vgl. Kap. 5.4). Mit der Bildung einer totalen, linearen Halbordnung der Zustandsänderungen im Tracemodell – unter Berücksichtigung von parallel ausgeführten Aktionen – wurde eine Vereinfachung eingeführt, die sich aus den unterschiedlichen Modellierungsmöglichkeiten von Parallelität im Metamodell ergibt (z.B. Nutzung von Threads vs. Rundlaufverfahren). Somit wurden in dieser Arbeit weitergehende Analysesprachen wie etwa CTL/CTL* nicht untersucht, eine weitergehende Forschung in diese Richtung unter speziellen Vorgaben zur Metamodellierung mit MActions scheint allerdings sinnvoll.

(b) *Wie können Eigenschaften wie Zustände, Parallelität und Zeit generisch in der Auswertung berücksichtigt werden?*:
Traces bilden die Schnittstelle zwischen operationaler Semantik und dynamischer Analyse, die semantisch nur über die Zustandsdefinition gekoppelt sind. Im Allgemeinen fallen Zustandsübergänge der Modellebene als *interne* Modellzustände für die Analyse (M1, Anwendungsebene) nicht notwendigerweise mit den „*Mikrozuständen*“ einer operationalen Semantik zusammen. Zeit oder Uhren als Zustandsgrößen im Laufzeitmodell bilden hier eine separate Ebene und sind nicht notwendigerweise deckungsgleich mit den Zustandstransitionen der operationalen Regeln (vgl. Kap. 4.1.2, 5.2.1). Die vielschichtigen Zustandsebenen von der Metamodellebene (M2) über die Modellebene (M1) hin zur Anwendungsebene, tragen maßgeblich zu jedwedem Ansatz einer dynamischen Analyse bei, so dass eine Konfiguration *unabhängig* von (bzw. nachträglich zu) der operationalen Semantik stattfinden sollte. Da die Zustandsgranularität über SGTs für MActions parametrisiert werden kann, lässt sich die Semantik der LT-OCL-Ausdrücke ändern und an die Bedürfnisse des Anwendungsmodells anpassen (vgl. Kap. 4.1.5). Aus dem parametrisierbaren Zustandsschema folgen gleich mehrere Äquivalenzklassen für einen Verhaltensvergleich, die sich anstatt über abstrakte Zustände und Aktionen über den vollständigen Vergleich von Laufzeitmodellen bilden lassen (vgl. Kap. 4.2.1).

(c) *Wo zeigen sich Parallelen zu existierenden Techniken und Kalkülen, die sich nicht im Kontext einer Metadatenhierarchie bewegen? Wo ergeben*

sich Vor- und Nachteile?:
Letztlich wurden existierende Konzepte zur prädikatenlogischen Temporallogik auf objektorientierte Strukturen der MOF-Metamodelle mittels einer Erweiterung von OCL übertragen. Die Basis bilden – wie für existierende Kalküle auch – Zustandssequenzen, jedoch wurde der bislang nicht vorhandene Bezug zur Semantikdefinition (und damit der Sprachunabhängigkeit durch Metamodelle) im Kontext von MOF erarbeitet (vgl. Kap. 4.1.3). Ferner erlaubt der prädikatenlogische Kern von LT-OCL und die Konzeption von Metamodell, Laufzeitmodell und logischer Instanziierung eine gezielte Spezifikation von zu überprüfenden dynamischen Ausdrücken in generischer Form (vgl. Kap. 4.3.3). Die Metadatenhierarchie hat den entscheidenden Vorteil, mit einheitlicher Strukturdefinition die Sprachunabhängigkeit zu ermöglichen. Die sonst selten in der Praxis zu findende Form einer prädikatenlogische Temporallogik spielt für den metamodellunabhängigen Ansatz die Hauptrolle. Als Nachteil kann die im Gegensatz zu formalen Kalkülen sperrige, graphische Notation und durch die auf MOF eingeschränkten Abstraktionsmöglichkeiten gesehen werden (vgl. Kap. 5.1.1). Weiterhin gestaltet sich die Navigation durch das Anwendungsmodell aufgrund der generischen Form oftmals schwierig (vgl. Kap. 4.4.1, A.4).

zu 3.: *Formalisierung der entwickelten Modellierungstechniken*: Die Brücke zwischen Metamodellierung und der dynamischen Analyse wurde durch mehrere Formalisierungsschichten erreicht (vgl. Kap. 2.4.2). Die mathematisch nicht formalisierten MOF-Modellierungskonzepte konnten unter Rückgriff und Adaption eines algebraischen Objektmodells beschrieben werden. Dazu wurden neben der Abbildung der Metamodellstruktur auf eine Algebra zwei äquivalente Sichten auf ein Metamodell gezeigt (Klassenstruktur des Metamodells vs. Instanzmodell, vgl. Kap. 2.3). Darauf aufbauend fügt ein um die OCL-Semantik ergänzter ASM die operationale Semantik der MActions mit der mathematischen Definition der OCL-Semantik zusammen (vgl. Kap. 21). Darüber hinaus wurde mittels der Axiomatisierung durch die Universelle Transformationsmaschine und der abstrakten Aktionssemantik der Rahmen für eine Reihe weiterer möglicher operationaler Semantiken gelegt, der über die Konzepte der MActions hinausgehen kann, aber die wesentlichen Eigenschaften der Modelländerungen für Ausführungsanalysen erhält (vgl. Kap. 2.5.4, 3.1.2).

(a) *Wie können statische und dynamische Aspekte von Metamodellen einheitlich mathematisch formalisiert werden?*: Die Beschreibung der dynamischen Aspekte erfolgt über Traces (Instanzen des Tracemetamodells), die sämtliche Änderungen am Laufzeitmodell als Sequenz erfassen. Durch Anwendung einer beliebigen Teilmenge dieser Sequenz lässt sich jeder durchlaufene Zustand als *induzierter* Zustand des Traces rekonstruieren und durch Abbildung auf das algebraische Objektmodell formal beschreiben (vgl. Kap. 4.1.4).

(b) *Ergeben sich Vorteile durch eine Nutzung von Metamodellen gegenüber formalen Kalkülen?*: Die Nutzung von Metamodellen zur Definition der abstrakten Syntax einer Sprache erlaubte bislang eine Ableitung von Werkzeugen und Editoren für die beschriebenen Modelle. Mit den durch MActions und LT-OCL gezeigten Erweiterungen in dieser Arbeit lassen sich darauf ad hoc Simulations- und Analysefunktionalitäten bereitstellen, die oftmals für andere Kalküle fehlen. Der sprachunabhängige An-

satz erlaubt eine direkte Erweiterung des jeweiligen Metamodells, ohne dass eine Reformalisierung in ein anderes Kalkül nötig würde. Zudem bilden Metamodelle über MOF eine strenge Hierarchie, die den Regeln der Instanziierung und Objekt-Metaklasse-Beziehung folgt, was für die Verarbeitung des Laufzeitmodells mittels MActions auch eine Integration und Wiederverwendung verschiedener Metamodelle ermöglicht (vgl. Kap. 5.1.2).

ANHANG A

Fallbeispiel: Programmiersprache C#

Der folgende Anhang beschreibt die Programmiersprache C# wie sie der ECMA 334 Standard spezifiziert [34]. Das Ziel dieser Fallstudie war es, eine in der Praxis etablierte Allzweckprogrammiersprache mit den Techniken der MActions zu beschreiben und exemplarisch mittels LT-OCL dynamische Analysen durchzuführen. Daher wurde weniger auf die vollständige Modellierung der Sprache C# fokussiert, als auf essentielle Konzepte w.z.B. Objekte, Felder, Methoden, Statements/Expressions sowie paralleler Ausführungsaspekte wie Threads (vgl. [20]).

A.1 C# Metamodell

Der ECMA Standard der Sprache enthält kein Metamodell, sondern spezifiziert die textuelle Syntax mittels einer EBNF Grammatik (vgl. Appendix A in [34]). Der erste Schritt in Richtung eines C# Metamodells wurde deshalb rein aus Sicht einer allgemein objektorientierten Sprache getätigt, ohne Anlehnung an die C# Grammatik. Dazu wurde ein initiales Metamodell mit OO-Basiskonzepten aufgesetzt, welches sich an die "Abstractions" der *UML Infrastructure* anlehnt (vgl. [35]). Dieses initiale Metamodell konnte anschließend anhand der grammatikalischen Sprachspezifikation verfeinert und um essentielle Teile ergänzt werden, darunter:

1. Typsystem mit Klassen, Primitive Typen, Arrays
2. Anweisungen und Ausdrücke (*Statements* und *Expressions*)
3. Laufzeitmodell und Nebenläufigkeit (Threading)

Der Kern des C# Metamodells beschreibt die strukturellen Aspekte der Sprache, d.h. Klassen, Schnittstellen, Methoden, Felder, etc. (s.Abb. A.1). Die Hauptvererbungshierarchie von Typen als Ableitungen der Klasse `CSType` ist neben den getypten Elementen (Ableitungen von `CSTypedElement`) charakteristisch für OO-Sprachen und entsprechend als Muster in der UML Infrastructure zu finden. Darunter finden sich dann die C# spezifischen Konstrukte wie `CSClass` mit `CSField` und `CSEvent` sowie die verhaltensbezogenen Klassen `CSMethod`, `CSConstructor`, `CSOperator` und `CSDelegate`. Weitere Syntaxvariationen ("syntactic suger") w.z.B. *property accessors* werden durch vereinheitlichte Klassen im Metamodell repräsentiert.

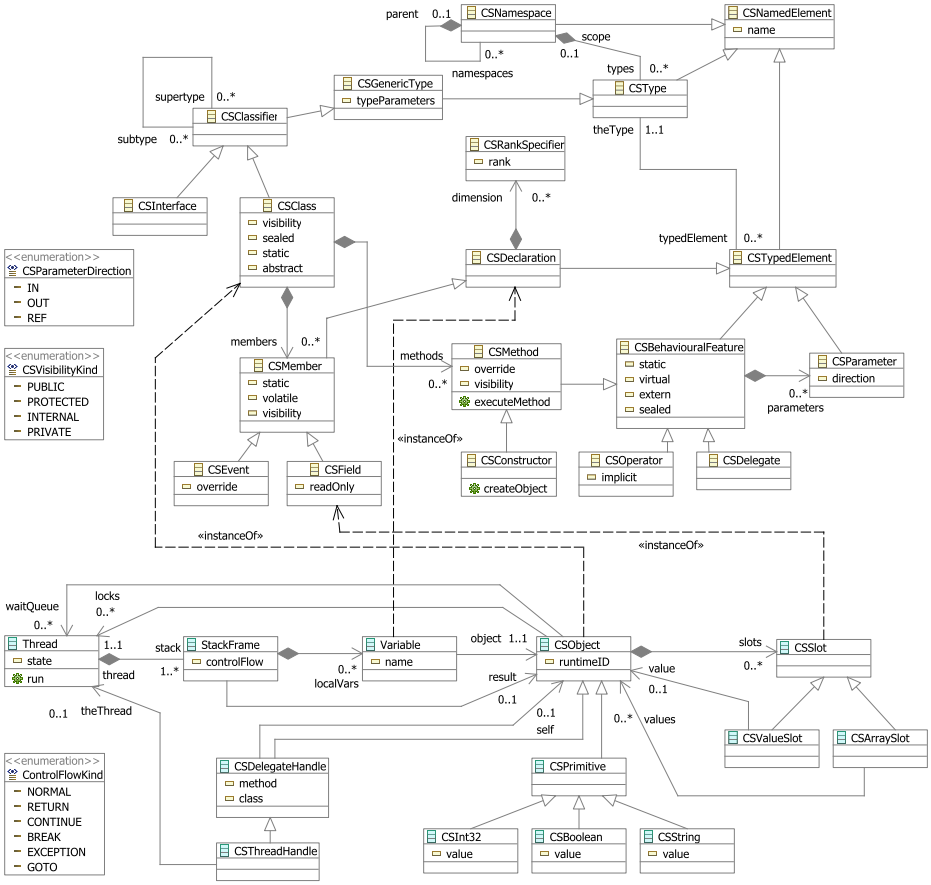


Abbildung A.1: Auszug aus dem C# Metamodell: strukturelle Sprachaspekte

Die Details zum Typsystem, Statements und Expressions sind zwar hier nicht dargestellt, sollen aber kurz erläutert werden, da die Definitionen eine für Programmiersprachen in der Praxis vielfach anwendbarem Muster folgen. Anweisungen und Ausdrücke bilden im Metamodell zwei zunächst unabhängige Vererbungshierarchien mit den Basisklassen `CSStatement` bzw. `CSExpression`. Unter diese Klassen gliedern sich dann Subklassen zu einzelnen Konstrukten wie `CSFor` oder `CSIfElse` für entsprechende Anweisungen, oder `CSInvoke` und `CSAssign` für Ausdrücke. Insgesamt 27 Klassen repräsentieren hierbei die Kontrollflusselemente und -struktur aller Anweisungen (also quasi die Zeilen eines Programms) und weitere 26 Klassen modellieren Ausdrücke, w.z.B. die Struktur von Aufrufen, Bedingungen, Variablenzugriffen und -zuweisungen, u.s.w. Im Prinzip sind diejenigen Konstrukte als eigenständige Metaklasse definiert, die in der Grammatik durch eine Expansionsregel beschrieben sind. Dazu wurden Gemeinsamkeiten in Klassen als "Zwischenabstraktionen" vereinheitlicht. Zum Beispiel bildet die Klasse `CSBinaryExp` die Basis für alle binären Ausdrücke wie `CSArithemticExp`, `CSRelationExp`, `CSAssignExp` und andere und enthält das gemeinsame Merkmal von zwei Containmentreferenzen `leftHand` und `rightHand` jeweils vom Typ `Expression`. Neben Referenzen wie die diese, welche beide Hierarchien zusammenfügen, existiert ein generelles `CSExpressionStatement`, über das ein Ausdruck zur Anweisung werden kann (z.B. einfacher Methodenaufruf). Dadurch baut sich die gesamte Sprachstruktur von `CSNamespace`, `CSClass`, `CSMethod` zu `CSStatement` und `CSExpression` mit Querreferenzierungen für Typen, Variablen- und Methodendeklarationen als Objektgraph auf.

A.2 Laufzeitmodell

Neben der abstrakten Syntax sind in Abb. A.1 die Klassen des C# Laufzeitmodells gezeigt, welche als Konfigurationen während der Simulation instanziiert und verändert werden. Die Klassen `CSObject` und `CSSlot` bilden auch hier die OO-typische Basis, gebunden durch die logische Instanziierung an die Definitionen der abstrakten Syntax. Eingebettet werden Objekte in den Kontext eines `Thread`, der über seinen Stapel (repräsentiert durch die Klasse `StackFrame`) Referenzen auf diese hält. Dazu werden lokale Variablen innerhalb einer Methoden jeweils als Instanz der Klasse `Variable` auf dem aktuellen `StackFrame` instanziiert.[1] Welche Variablen wo und wann angelegt und manipuliert werden, bestimmen letztlich wie oben beschrieben Deklarationen und variablenbezogene Ausdrücke, modelliert durch `CSVarDeclaration` und `CSVarExp` (beide nicht dargestellt).

Als C# Spezifikum bieten *Delegates* eine zusätzliche Möglichkeit, einen sog. Delegattyp zu definieren, der eine Methodensignatur als quasi Funktionstyp spezifiziert. Dadurch lassen sich z.B. Methoden als Parameter übergeben. Im Metamodell ist diese Eigenschaft durch die Klasse `CSDelegateHandle` modelliert, wobei die Objektbindung zum Zeitpunkt der Delegateninstanziierung durch die Attribute `method` und `class` deklariert ist und mittels `self` einem Delegatobjekt zugeordnet wird.

Im Hinblick auf Threads verweist der C# ECMA Standard auf die *Common Language Infrastructure* (CLI), welche eine für verschiedene Hochsprachen vereinheitliche Laufzeitumgebung beschreibt und als Implementierung in der Microsoft .NET Plattform verwirklicht wurde [58]).[2] Threading ist in der Standardbibliothek im Namensraum `System.Threading` realisiert und durch Objekte der Bibliotheksklasse `Thread` im Programm widergespiegelt. Hierzu muss i.Allg. durch ein `ThreadStart` Delegate der Einstiegspunkt spezifiziert und an den Konstruktor der Klasse Thread übergeben werden. Zur speziellen Handhabung in der Verhaltensbeschreibung dient die Klasse `CSThreadHandle`. Neben den delegattypischen Eigenschaften verwaltet sie einen direkten Verweis auf den ausführenden Thread via `theThread`.

A.3 C# Semantik

Anhand der vollständig im Metamodell erfassten abstrakten Syntax der Sprache C#, wurde für einen Großteil die Semantik in MActions definiert. Diese Teilmenge umfasst im einzelnen die folgenden Sprachkonzepte, die hier nur als Zusammenfassung aufgeführt werden sollen:

Typen und OO-Konzepte: Klassen, Methoden, Felder, Konstruktoren, Methodenaufrufe und -auflösung, Parameterübergabe mit Referenz- und Wertparametern, Delegattypen, primitive Typen `bool`, `int`, `decimal`, `string`, `enum`, Arrays

Anweisungen: `for` und `while`-Schleifen, `if-then-else`, Variablendeklarationen, Zuweisungen, Kontrollflussauswertung mit `return`, `break` und `continue`

Ausdrücke: `new`, arithmetische und boolesche Ausdrücke, relationale Vergleichsoperatoren, Feld- und Array-Zugriff, Variableninitialisierung, Literale

Zusätzlich wurden von der Standardbibliothek die essentiellen Ein- und Ausgabemethoden (d.h. `Console`) sowie Thread bezogene Methoden modelliert. Ohne die

[1] Weiter geschachtelte *Scopes*, d.h. {...} Blöcke innerhalb von Methoden, wurden nicht modelliert

[2] Tatsächlich wird wiederum auf ein XML Begleitdokument verwiesen, welches Die Schnittstelle der Standardbibliothek beschreibt

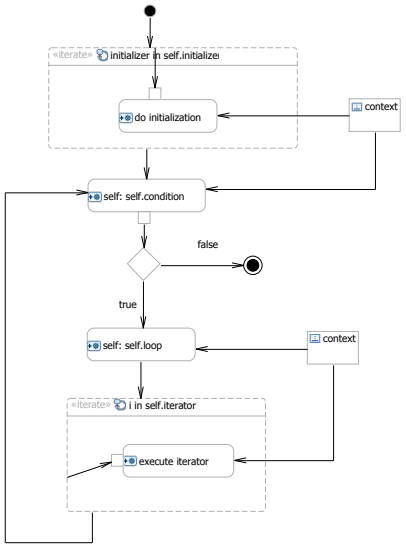
MAction-Definitionen hier im Detail vollständig aufzulisten, sollen einige Definitionen exemplarisch hervorgehoben und diskutiert werden.

In der C# MActions Semantik sind die Aktionen überwiegend direkt den Klassen der abstrakten Syntax zugeordnet. Alle Sprachelemente die Verhalten ausdrücken, erhielten eine Haupt-MOperation, die die Semantik spezifiziert und ggf. durch weitere Operationen unterstützt wird. Zum Beispiel beschreibt `CSMethod-#executeMethod`, wie der Rumpf einer Methode Anweisung für Anweisung abgearbeitet wird, `CSConstructor#createObject` wie Objekte instanziiert werden oder `CSFor#executeFor()`, wie zunächst die Schleifeninitialisierung, dann die Schleifenbedingung und anschließend die Schleife selbst ausgeführt werden (s.Abb. A.2(a)). Dabei wurde sich die Vererbungsbeziehung der Statement- und Expressionklassen zu Nutze gemacht, um eine abstrakte MOperation der Basisklassen `CSStatement` bzw. `CSExpression` in allen Subklassen dieser Vererbungshierarchien gezielt mit der Semantik jedes einzelnen Konstrukts zu überschreiben. Dieses Vorgehen ist angelehnt an das Interpreter Entwurfsmuster (vgl. [112]). Dadurch kann ein Großteil der Ausführungssemantik durch Aufruf der Basisoperation modelliert werden, wobei die Überschreibung bei der Ausführung mittels Polymorphie effektiv direkt zum Aufruf des gewünschten spezielleren Verhaltens führt.

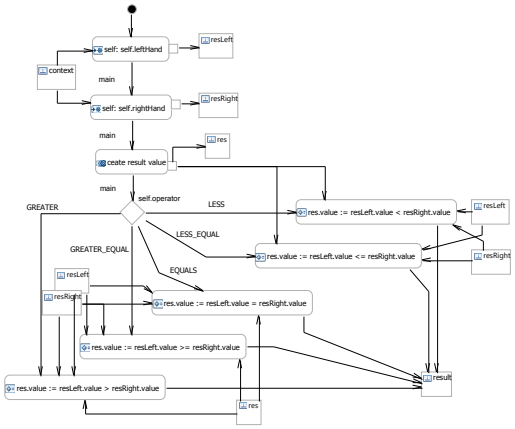
Bei der zuletzt gezeigten `for`-Schleifendefinition ist gut zu erkennen, wie die Ausführungssemantik in Aktionen überführt werden konnte, die unmittelbar einer Formalisierung der Ablaufbeschreibung entspricht. Hinter den Invokationsaktionen `do initialization`, `self.condition` (Auswertung der Schleifenbedingung), `self.loop` sowie `execute iterator` verbergen sich Aufrufe an die *evaluate* Operation der Klasse `CSExpression`, welche durch besagte Methodenüberladung letztlich zur Ausführung der spezifischen Semantikoperationen führt. Das selbe Prinzip kommt bei fast allen Semantikbeschreibungen von C#-Ausdrücken zum Tragen, z.B. bei den in Abb. A.2(b) dargestellten `CSRelationExp`. Als binärer Ausdruck mit linker und rechter Seite (d.h. Teilausdrücke vor und nach dem Relationsoperator), findet zunächst eine Auswertung dieser Teilausdrücke statt (`self.leftHand` bzw. `self.rightHand`). Anschließend wird das Ergebnisobjekt erzeugt und mittels einer MDecisionNode über `self.operator` zwischen den jeweiligen Vergleichsoperatoren unterschieden. Insgesamt gleicht dieses Vorgehen zur Beschreibung der Semantik dem typischen Interpreter Entwurfsmuster über einen abstrakten Syntaxbaum, dass mittels MActions direkt im Metamodell umgesetzt werden konnte.

Ein weiterer wichtiger Aspekt ist die Zustandsverwaltung und das Ändern von Werten. Zugriff auf den C# Stapel und somit auf den Ausführungskontext bietet i.Allg. ein Operationsparameter `context : StackFrame`, der bei allen MOperationen durchgeschleift wird und eine Referenz auf das aktuelle StackFrame darstellt. Dadurch können Ausdrücke und Anweisungen kontextspezifisch operieren und z.B. Methodeninkarnationen alle notwendigen Daten auf dem Stapel ablegen (Parameter, Variablen, Rückgabewerte, Threadinformationen). Abbildung A.2(c) zeigt die Semantik von Variablen- und Feldzuweisungen. Abhängig vom Typ des Ausdrucks der auf der linken Seite steht (Referenz `leftHand` [3]), wird bei der Zuweisung zunächst zwischen Variablen und Feldern unterschieden. Bei letzteren dann noch weiter zwischen einfachen Feldern und Arrays. Den Einstieg ermöglichen Abfragen von `frame.localVars`, die über den Namen z.B. die zu aktualisierende Variable oder das zu aktualisierende Feld vom Stapel holen und in nachfolgender MAssign-Aktion den durch `rightHand`-Ausdruck ermittelten Wert zuweisen. Mittels logischer

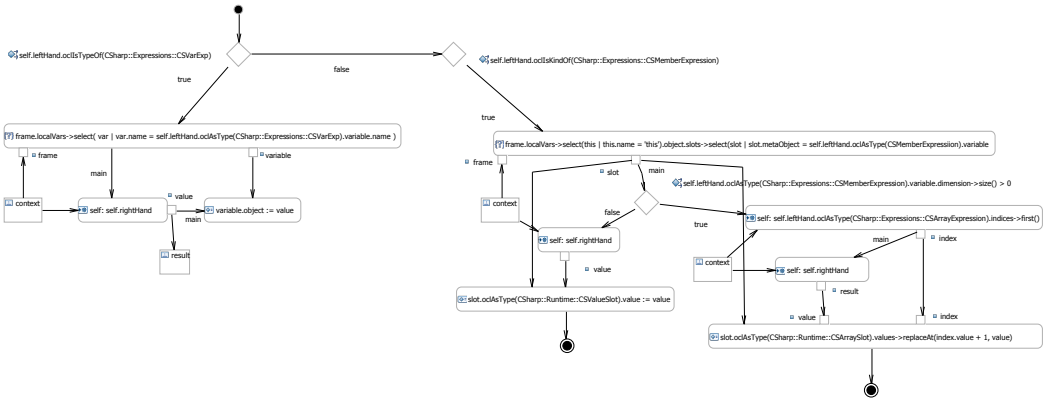
[3] also dem sog. "l-value" als Ziel einer Zuweisung im Gegensatz zu "r-values" auf der rechten Seite die einen Wert liefern



(a) `for`-Schleife



(b) Auswertung relationaler Vergleiche

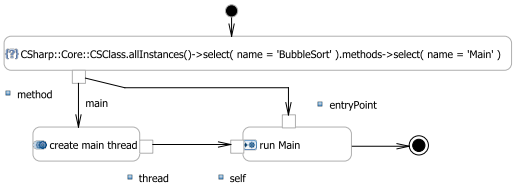


(c) Variablenzuweisung, Feldzuweisungen

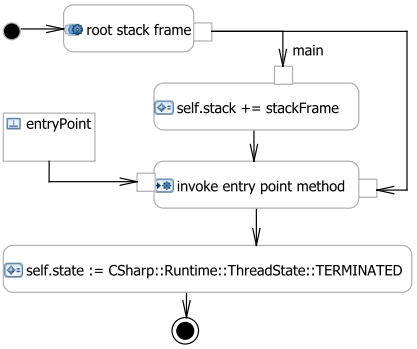
Instanziierung kann für Felder durch die `metaObject` Referenz gezielt der Slot erfragt werden, der Werte für das Feld enthält. Das aktuelle C# Objekt wird für Feldzugriffe über eine spezielle `this` Variable auf dem Stapel repräsentiert und ist bei jedem Methodenaufruf ein impliziter Parameter. Die hier nicht dargestellte Aktionssemantik von `CSInvoke` stellt das Ablegen dieser Referenz auf dem StackFrame sicher. Bei Arrays muss darüber hinaus der Indexausdruck ausgewertet werden, um die Zielposition im Array zu ermitteln.

Die Ausführung eines C# Programms ist beschrieben durch das Threadmodell der CLI. Neben der sequentiellen Programmausführung des "Main-Threads", können weitere Threads über die Standardbibliothek instanziiert und ausgeführt werden. Dabei erhält jeder Thread seinen eigenen Stapel.

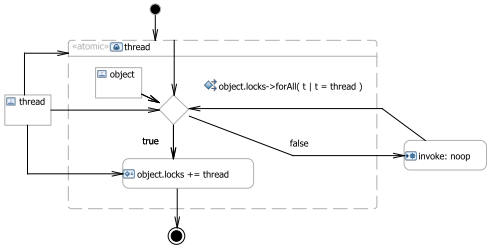
Der globale Programmeinstiegspunkt ist vereinfacht modelliert als MActivity `ExecuteClass` (s.Abb. A.2(d)). Diese spezifiziert den Programmstart und wird bei der MActions Ausführungsumgebung als Einstiegspunkt angemeldet. Als erste Aktion wird im Syntaxraum nach einer festgeschriebenen Methode `Main` in einer C# Klasse namens "BubbleSort" gesucht. Diese fixe Suche durch alle Instanzen von `CSClass` vereinfacht den Simulationseinstieg und ersetzt eine Auswertung von globalen Objekten von z.B. Kommandozeilenparametern, die in die Simulation hineingereicht werden könnten. Für unsere Experimente wurde die Klasse entsprechend der Beispiele aus Abschnitt A.4 jeweils angepasst. Das Resultat, welches am Ausgabepin `method` anliegt, wird anschließend nach erzeugen des Main-Threads als `entryPoint` Parameter übergeben und ausgeführt (vgl. Abb. A.2(e)).



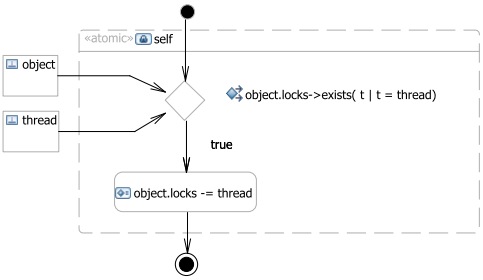
(d) ExecuteClass



(e) Thread#run



(f) AcquireLock



(g) ReleaseLock

Abbildung A.2: Threading der CLI

Weitere Threads können wie einleitend beschrieben durch ThreadStart-Delegat erzeugt werden. Die Sprachsemantik nutzt zur Modellierung MThreads der MActions um die Konzepte abzubilden. Das bedeutet, dass für jeden C# Thread faktisch

ein MThread gestartet wird. Dies geschieht in der Modellierung der ThreadStart-Semantik durch Erzeugen einer neuen C# `Thread`-Instanz mit anschließendem Aufruf der `run`-MOperation mit `startThread = true` (vgl. Abschnitt 3.3.3). Andere Modellierungen sind denkbar, für eine Diskussion der Thematik vergleiche auch Abschnitt 2.5.3 und 5.2.2.

C# Methoden und Anweisungen sind per se nicht thread-safe (s. Abschnitt 10.10 in [34]). Zu diesem Zweck existiert die `lock` Anweisung, mit deren Hilfe kritische Bereiche über ein Objekt geschützt und synchronisiert werden können. Die (Ent-)-Sperrung von Objekten erfolgt somit explizit durch `lock`, die zu Semaphoren, Monitoren und so weiter in der Standardbibliothek ausgebaut sind. Entsprechende Modellierung der Kernlogik zum Akquirieren und Freigeben eines Objekts ist in Abbildungen A.2(f) und A.2(g) gezeigt. Zur Referenzierung des in einen kritischen Bereich eingetretenen Threads existiert im Laufzeitmodell die Referenz `locks`, zur Verwaltung der Warteschlange weiterer Threads die Liste `waitQueue`. Sollte ein Thread warten müssen, so wird dieser durch eine `noop` in eine aktive Warteschleife versetzt. Da bei der Ausführung MThreads zum Einsatz kommen, sind diese wiederum *selbst* gegenseitig durch eine MAtomicGroup synchronisiert.

A.4 Beispielprogramme und -analysen

Für eine exemplarische Analyse wurden repräsentative Beispiele gewählt, darunter eine rekursive Funktion (Fakultätsfunktion), Objektstrukturen, ein Sortieralgorithmus (BubbleSort) sowie einige Experimente zum Threading. Jedes Beispiel bringt recht typische Fragestellungen über dynamisches Verhalten mit sich, die auch in anderen Kontexten eine Rolle spielen. Als Auswahl sollen im Folgenden zwei Analysen vorgestellt werden.

A.4.1 Rekursive Funktionen

Es wurde die Fakultätsfunktion wie in Listing A.1 gezeigt spezifiziert. Kern ist die rekursive Funktion `f`, die als Methode in der Klasse `Algorithms` definiert ist. Wie in Abschnitt A.3 beschrieben wird immer die Methode namens `Main` als Einstiegspunkts in ein Programm aufgerufen. Somit wird hier wie gezeigt nach anlegen eines Objekts `a` die Methode mit einem konstanten Wert aufgerufen und das Ergebnis ausgegeben.

```
1  class Algorithms {
2    public Algorithms() {}
3
4    public int f( int n ) {
5      if (n == 0)
6        return 1;
7      else
8        return n * f(n-1);
9    }
10
11   static void Main(string[] args) {
12     Algorithms a = new Algorithms();
13     Console.WriteLine(a.f(5));
14   }
15 }
```

Listing A.1: Fakultätsfunktion, rekursiv

Mit Blick auf die Zustände ergibt sich mit dem Laufzeitmodell aus Abschnitt A.2 eine recht einfache Struktur, zunächst bestehend aus einem Thread mit Stackframe für `Main`, in das anschließend durch die Variablendefinition `a` eine Variable in `localVars` hinzugefügt wird. Der eigentlich spannende Teil ist das rekursive Aufbauen von Stackframes durch den Aufruf von `f`. Wie zu erwarten wächst der Stapel mit jeder Methodeninkarnation um genau ein Element.

Eine dynamische Analyse eines C# Programms könnte sich den Stapel als Zielgröße setzen und fragen, ob er innerhalb eines bestimmten Limits *stackLim* bleibt oder es überschreitet.[4] Eine LT-OCL-Bedingung, die dieses temporal logisch für jeden Thread eines C#-Programms formuliert ist:

```
context Thread inv:
always(self.stack->size() < stackLim)
```

Diese allgemeine Laufzeitinvariante läßt sich für ausgewählte Funktionen spezialisieren und zur Spezifikation eines Terminierungskriteriums nutzen. Zum Beispiel könnte man für die Fakultätsfunktion als rekursive Funktion fordern, dass bei jeder Inkarnation der Wert aller Parameter jeweils echt kleiner ist als bei vorheriger Inkarnation. Durch den Zugriff auf das Laufzeitmodell mit Stapel und Variablen wird dies ermöglicht:

```
let stEl : self.stack->at(self.stack->size()-1) in
def previousStackValue(v : Variable) : Integer =
  if (stEl.oclIsUndefined()) then maxInt()
  else stEl->localVars->select(name = v.name).object.value.toInteger()
  endif

context Thread inv:
  always(self.stack->last().localVars->forAll(var |
    var.object.value.toInteger() < previousStackValue(var)))
```

Die Schwierigkeit ergibt sich generell bei der Einschränkung auf den gewünschten Objektkontext (hier die konkrete Funktion f), welcher im gezeigten LT-OCL-Ausdruck vernachlässigt wurde. Das macht das Formulieren von z.B. Schleifeninvarianten zu einem Problem.

A.4.2 Algorithmen

Ähnliche Laufzeitbedingungen lassen sich oftmals für Algorithmen formulieren. Als Beispiel wählen wir den bekannten Sortieralgorithmus *BubbleSort*, der von der Kommandozeile Zahlen einliest, diese in ein Array speichert, aufsteigend sortiert und am Ende ausgibt:

```
1  class BubbleSort {
2    private int[] numbers;
3
4    public BubbleSort() {
5      numbers = new int[10];
6      for( int i = 0; i < 10; i++) {
7        numbers[i] = Convert.ToInt32(Console.ReadLine());
8      }
9    }
10
11   public void bubblesort() {
```

[4] Natürlich ist in diesem einfachen Fall von linearer Rekursion die Antwort trivial, für komplexere Formen von verschachtelten oder wechselseitige rekursive Funktionen gilt dies aber nicht.

```
12      for ( int i = 0; i < numbers.Length; i++ ) {
13        for ( int j = 0; j < numbers.Length − i − 1; j++ ) {
14          if (numbers[j] > numbers[j+1]) {
15            swap(j, j+1);
16          }
17        }
18      }
19    }
20
21    private void swap(int i, int j) {
22      int temp = numbers[i];
23      numbers[i] = numbers[j];
24      numbers[j] = temp;
25    }
26
27    public void print() {
28      for (int i = 0; i < numbers.Length; i++) {
29        Console.WriteLine(numbers[i]);
30      }
31    }
32
33    static void Main(string[] args) {
34      BubbleSort sorter = new BubbleSort();
35      sorter.bubblesort();
36      sorter.print();
37    }
38  }
```

Listing A.2: Sortieralgorithmus BubbleSort

Von einem Sortieralgorithmus erwartet man, dass am Ende der Ausführung alle Zahlen in die richtige (aufsteigende oder absteigende) Reihenfolge gebracht worden sind:

```
let numbers : CSArraySlot = CSArraySlot.allInstances()−>asSequence()−>first()
in
  eventually(numbers.values−>forAll(a, b |
    numbers.values−>indexOf(a) < numbers.values−>indexOf(b) implies a <= b))
```

Eine komplexere Form einer Schleifeninvariante könnte man für den Algorithmus auch über den Slot `numbers` spezifizieren. Die äußere Schleifeninvariante für BubbleSort, also dass die Werte $numbers[i+1]$ bis $numbers[9]$ sortiert sind, ließe sich etwa wie folgt formulieren:

```
context StackFrame inv:
let execBubblesort : Boolean = self.pc.eContainer().oclIsTypeOf(CSMethod) and
  self.pc.eContainer().oclAsType(CSMethod).name = 'bubblesort',
  i : Variable = self.localVars−>select(name = 'i')
in
always(execBubblesort implies
   numbers.values−>forAll(a, b |
    (numbers.values−>indexOf(a) > i.value and numbers.values−>indexOf(b) > i.value)
      implies (numbers.values−>indexOf(a) < numbers.values−>indexOf(b)
        implies a <= b)))
```

An diesem Beispiel zeigt sich recht gut, dass die eigentliche Bedingung recht kompakt zu formulieren ist, die Einschränkung auf den Kontext (d.h. die Funktion `bubblesort`) aber schwieriger. Sollte zum Beispiel ein konkretes Objekt adressiert

werden, müsste man den Einstieg über eine Variable suchen, die auf das Objekt verweist und anschließend den Kontext sukzessive weiter einschränken. Ganz ähnlich gelagert ist der Fall für die Invariante der inneren Schleife. Dazu wäre eine Einschränkung der LT-OCL-Bedingung auf die Position der inneren `for`-Schleife nötig, was sich letztlich nur über die Position des `CSStatement` erreichen ließe.

Literaturverzeichnis

[1] OMG. Model driven architecture guide, version 1.0.1. June 2003. omg/03-06-01.

[2] Thomas Stahl und Markus Völter. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management.* Dpunkt Verlag, 2005.

[3] OMG. *UML 2.0 Superstructure Specification.* Object Management Group, October 2004. ptc/04-10-02.

[4] Martin Fowler. *DSL: An Introductory Example.* http://martinfowler.com/dslwip/Intro.html. Last checked: . Januar 2009.

[5] Olaf Kath, Michael Soden, Marc Born, Tom Ritter, Andrej Blazarenas, M. Funabashi, and C. Hirai. An open modeling infrastructure integrating EDOC and CCM. In *EDOC '03: Proceedings of the 7th International Conference on Enterprise Distributed Object Computing*, page 198, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Marc Born, Ina Schieferdecker, Hans-Gerhard Gross, and Pedro Santos. Model-driven development and testing — a case study. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, University of Twente, Enschede, the Netherlands, 2004.

[7] Luigi Mazzucchelli. Visualization and distributed systems technologies: The ad4 approach and beyond. In *Proceedings of the Final AD4 Workshop on Visualization and Distributed Systems Technologies*, Rome, 2007.

[8] OMG. *Meta Object Facility (MOF) 2.0 Core Specification.* Object Management Group, October 2003. ptc/03-10-04.

[9] Rainer Ehre and Malte Kaufmann. The SAP eclipse story, 11 2007.

[10] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context - Motorola case study. In *Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, October 2-7*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491, Montego Bay, Jamaica, 2005. Springer.

[11] Michael Sommerhalder. Design guidelines und erfahrungsberichte in MDD/MDA. In *Seminar für Model-Driven Software Development*, Institut für Informatik der Universität Zürich, 2007.

[12] Joachim Hössler, Olaf Kath, Michael Soden, Marc Born, and S. Saito. Significant productivity enhancement through model driven techniques: A success story. In *EDOC '06: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference*, pages 367–373, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Hartmut Bossel. *Modellbildung und Simulation: Konzepte, Verfahren und Modelle zum Verhalten dynamischer Systeme.* Vieweg Verlagsgesellschaft; Auflage: 2., veränd. Aufl. 1994.

[14] Sebastian Günther. Die sprachbestandteile von doänenspezifischen sprachen: Eine ableitung aus sprachphilosophischen un linguistischen wurzeln der informatik, 2008.

[15] Terence Parr. *ANTLR – Another tool for language recognition.* Last checked: . Februar 2006.

[16] Roswitha Bardohl. *GENGED: Visual Definition of Visual Languages based on Algebraic Graph Transformation.* PhD thesis, Technische Universität Berlin, December 1999.

[17] OMG. *OCL 2.0 Specification.* Object Management Group, May 2006. formal/2006-05-01.

[18] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Draft adopted Specification, ptc/05-10-02.* Object Management Group, 2005.

[19] Joachim Fischer, Michael Piefel, and Markus Scheidgen. A metamodel for SDL-2000 in the context of metamodelling ULF. In Daniel Amyot and Alan W. Williams, editors, *System Analysis and Modeling: 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1–4, 2004, Revised Selected Papers*, volume 3319 / 2005 of *Lecture Notes in Computer Science*, page 208. Springer-Verlag GmbH, 2005.

[20] Michael Soden and Hajo Eichler. An approach to use executable models for testing. In *Enterprise Modelling and Information Systems Architectures - Concepts and Applications , Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures*, volume P-119 of *LNI*. GI, 2007.

[21] G.D. Plotkin. A structural approach to operational semantics. Technical report, University of Aarhus, Denmark, 1981.

[22] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M.Sintzoff, C.H. Lindsey, L.G.T. Meertens, and R.G.Fisker. Revised report on the algorithmic language ALGOL 60. *Comm. ACM*, 6:1–20, 1963.

[23] Charles Wallace. The semantics of the C++ programming language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.

[24] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of java. Technical report, 1999.

[25] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of c#. In *Journal Theoretical Computer Science*, 2004.

[26] Andreas Prinz. *Formal Semantics for RSDL: Definition and Implementation.* PhD thesis, Humboldt-Universität zu Berlin, June 2000.

[27] Egon Börger, Alessandra Cavarra, and Elvinia Riccobene. An ASM semantics for UML activity diagrams. In *AMAST '00: Proceedings of the 8th International Conference on Algebraic Methodology and Software Technology*, pages 293–308, London, UK, 2000. Springer-Verlag.

[28] Sabine Kuske. A formal semantics of uml state machines based on structured graph transformation. In *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools, volume 2185 of LNCS*, pages 241–256. Springer, 2001.

[29] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[30] Yuri Gurevich. Abstract state machines: An overview of the project. Technical report, Microsoft Research, 2003.

[31] Jim Huggins. *Abstract State Machine.* Last checked: . April 2009.

[32] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. Technical report, Microsoft Research, 2004.

[33] Programming languages — C. ISO/IEC 9899:1999.

[34] C# language specification. Standard ECMA-334, European association for standardizing information and communication systems, 2006.

[35] OMG. *UML 2.0 Infrastructure Specification.* Object Management Group, September 2003. ptc/03-09-15.

[36] C.H.A. Koster. A shorter history of Algol68. http://npt.cc.rsu.ru/user/wanderer/ODP/ALGOL68.txt.

[37] Peter D. Mosses. *Action semantics.* Cambridge University Press, New York, NY, USA, 1992.

[38] T. Team. *Triskell Meta-Modelling Kernel. IRISA, INRIA. www.kermeta.org.*

[39] CETEVA. XMF. http://itcentre.tvu.ac.uk/ clark/xmf.html.

[40] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. Atl: a qvt-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720, New York, NY, USA, 2006. ACM.

[41] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic anchoring with model transformations. In Alan Hartman and David Kreische, editors, *Model Driven Architecture - Foundations and Applications: First European Conference*, pages pp. 115 – 129. Springer, November 2005.

[42] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson.

[43] *Introduction to Simulation*, 1995.

[44] Paul A.Fishwick. *Simulation Model Design and Execution: Building Digital Worlds*. International Series in Industrial and Systems Engineering. Prentice-Hall, 1995.

[45] The MathWorks. MATLAB & SIMULINK suite. http://www.mathworks.de/.

[46] ESTEREL Technologies. SCADE suite. http://www.esterel-technologies.com/products/scade-suite/.

[47] PragmaDev. PragmaDev real time developer studio. http://www.pragmadev.com/.

[48] Robert E. Shannon. Introduction to simulation languages. In *WSC '77: Proceedings of the 9th conference on Winter simulation*, pages 14–20. Winter Simulation Conference, 1977.

[49] Robert E. Shannon. Introduction to the art and science of simulation. In *WSC '98: Proceedings of the 30th conference on Winter simulation*, pages 7–14, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[50] Andrew Watson. UML vs. DSLs: A false dichotomy. April 2007. omg/08-09-03.

[51] SIMULA Standards Group. SIMULA standard. http://prosjekt.ring.hibu.no/simula/Standard/index.html, August 1986.

[52] Jerry Banks (Editor) et al. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. Engineering & Managenment Press, Hardcover, September 1998.

[53] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3. edition, 1997.

[54] ODEMx — C++ class library for process simulation. http://odemx.sourceforge.net/.

[55] Joachim Fischer and Klaus Ahrens. *Objektorientierte Prozeßsimulation in C++*. Addison Wesley, 1996.

[56] VDI-Gesellschaft Fördertechnik Materialfluss-Logistik. *VDI 3633 — Simulation von Logistik-, Materialfluß- und Produktionssystemen*. Beuth Verlag.

[57] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification Third Edition*. Addison Wesley, 2005.

[58] Common Language Infrastructure (CLI). Standard ECMA-335, European association for standardizing information and communication systems, 2006.

[59] Thomas Ball. The concept of dynamic analysis. In *Proc. 7th European Software Engineering Conference (ESEC'99)*, 1999.

[60] IEEE Standards Board. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, Dec 1990.

[61] Thomas Bauer, Hajo Eichler, Axel Rennoch, and Sebastian Wieczorek (Eds.). Model-based testing in practice, June 2009.

[62] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental evaluation of verification and validation tools on martian rover software, 2003.

[63] Oleg Sokolsky, Klaus Havelund, and Insup Lee. Introduction to the special section on runtime verification. *STTT*, 14(3):243–247, 2012.

[64] Diego Lo Giudice, David Metcalfe, Mike Gilpin, Matthew McCormack, and Megan Daniels. The state of model-driven development. Report on MDA acceptance, http://www.forrester.com, April 2007. Forrester Research Inc.

[65] Miroslaw Staron. Adopting MDD in industry - a case study at two companies. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems MoDELS 2006*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. ACM/IEEE, Springer, 2006.

[66] Klaus Krogmann and Steffen Becker. A case study on model-driven and conventional software development: The palladio editor. In *Software Engineering (Workshops)*, pages 169–176, 2007.

[67] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. 2006.

[68] Joachim Hößler and Michael Soden. OCL support in MOF repositories. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, March 17-18*, 2004.

[69] Joachim Hößler, Hajo Eichler, and Michael Soden. Coevolution of models, metamodels and transformations. *Models and Human Reasoning*, June 2005.

[70] Michael Soden, Hajo Eichler, and Joachim Hößler. Inside MDA: Mapping MOF2.0 models to components. In *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, March 17-18*, 2004.

[71] Markus Scheidgen. *Describing Computer Languages: Meta-languages to describe languages, and meta-tools to automatically create language tools.* PhD thesis, Humboldt-Universität zu Berlin, August 2008.

[72] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework (The Eclipse Series).* Addison-Wesley Professional, first edition, August 2003.

[73] NetBeans / Sun Microsystems. *Netbeans Meta Data Repository (MDR).* Last checked: . Januar 1970.

[74] Markus Scheidgen. *A MOF 2.0 for Java.* Last checked: . Februar 2006.

[75] Symposium on Eclipse open source software and OMG open specifications. www.omg.org/eclipse-omg/, 2008.

[76] OMG. *Meta Object Facility, Version 1.4*. Object Management Group, March 2003. formal/2002-04-03.

[77] Markus Scheidgen. CMOF-model semantics and language mapping for MOF 2.0 implementation. In *Joint Meeting of the 4th Workshop on Model-Based Development of Computer Based Systems (MBD) and 3rd International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), 13th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, 2006.

[78] OMG. *MOF2.0/XMI Mapping Specification, v2.1*. Object Management Group, sept 2005. formal/2005-09-01.

[79] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *UML'01: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, LNCS, pages 19–33, London, UK, 2001. Springer-Verlag.

[80] Colin Atkinson and Thomas Kühne. Rearchitecting the uml infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[81] Artur Boronat and José Meseguer. An algebraic semantics for MOF. In *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, March 29-April 6,*, volume 4961 of *Lecture Notes in Computer Science*, pages 377–391, Budapest, Hungary, 2008. Springer.

[82] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *Monographs in Theoretical Computer Science. An EATCS Series*. Springer, 1985.

[83] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[84] Dana Scott and Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. Technical Report PRG-6, 1971.

[85] Mark Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[86] J. McCarthy. Towards a mathematical science of computation. In *In IFIP Congress*, pages 21–28. North-Holland, 1962.

[87] Gordon D. Plotkin. The origins of structural operational semantics. In *Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[88] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[89] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[90] V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific, 1995.

[91] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[92] Karl-Heinz Buth. Simulation of SOS definitions with term rewriting systems. In *ESOP*, pages 150–164, 1994.

[93] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 229–239, New York, NY, USA, 1988.

[94] Luca Aceto. GSOS and finite labelled transition systems. *Theor. Comput. Sci.*, 131(1):181–195, August 1994.

[95] Luca Aceto, Wan Fokkink, and Chris Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, pages 197–292. Elsevier, 1999.

[96] Bard Bloom. Ready simulation, bisimulation, and the semantics of CCS-like languages, 1993.

[97] Yuri Gurevich. Evolving algebras 1993: Lipari guide. pages 9–36, 1993.

[98] Egon Börger, Angelo Gargantini, and Elvinia Riccobene, editors. *Abstract State Machines, Advances in Theory and Practice, 10th International Workshop, ASM 2003, Taormina, Italy, March 3-7, 2003, Proceedings*, volume 2589 of *Lecture Notes in Computer Science*. Springer, 2003.

[99] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, UK, 1981. Springer-Verlag.

[100] Robin Milner. *Communicating and mobile systems: the Pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[101] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, May 1994.

[102] Yuri Gurevich and Dean Rosenzweig. Partially ordered runs: A case study. In *Proceedings of the International Workshop on Abstract State Machines, Theory and Applications*, ASM '00, pages 131–150, London, UK, UK, 2000. Springer-Verlag.

[103] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 4:578–651, 2003.

[104] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms: Correction and extension. *ACM Transactions on Computational Logic*, 9(3):1–32, 2008.

[105] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[106] S. Abiteboul, L. Herr, and J. van den Bussche. Temporal versus first-order logic in query temporal databases. In *ACM Symposium on Principles of Database Systems*, pages 49–57, Montreal, Canada, 1996.

[107] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8:244–263, 1986.

[108] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[109] César Sánchez and Martin Leucker. Regular linear temporal logic with past. In *VMCAI*, pages 295–311, 2010.

[110] Robert Goldblatt. Mathematical modal logic: a view of its evolution. *J. of Applied Logic*, 1(5-6):309–392, 2003.

[111] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[112] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[113] Markus Scheidgen. Textual modelling embedded into graphical modelling. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 153–168, Berlin, Heidelberg, 2008. Springer-Verlag.

[114] Eclipse. *Graphical Modeling Framework, http://www.eclipse.org/gmf*. Last checked: . Januar 2009.

[115] Obeo. *Obeo Designer, http://www.obeodesigner.com/*. Last checked: . Juni 2011.

[116] Andreas Blunk. MODEF ein generisches debugging-framework für domänenspezifische sprachen mit metamodellbasierter sprachdefinition auf der basis von eclipse, emf und eprovide, May 2009.

[117] Matthias Meyer and Lothar Wendehals. Selective tracing for dynamic analyses. In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proc. of the 1st Workshop on Program Comprehension through Dynamic Analysis (PCODA), co-located with the 12th WCRE, Pittsburgh, Pennsylvania, USA*, volume 2005-12 of *Technical Report*, pages 33–37. Universiteit Antwerpen, Belgium, November 2005.

[118] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[119] Hajo Eichler and Michael Soden. An approach to behaviour comparison using execution traces (whitepaper). 2008.

[120] Michael Soden and Hajo Eichler. Temporal extensions of OCL revisited. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 190–205, Berlin, Heidelberg, 2009. Springer-Verlag.

[121] Markus Scheidgen and Joachim Fischer. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 157–171, Haifa, Israel, 2007. Springer.

[122] Ian M. Hodkinson, Frank Wolter, and Michael Zakharyaschev. Decidable fragment of first-order temporal logics. *Annals of Pure and Applied Logic*, 106(1-3):85–134, 2000.

[123] Hajo Eichler, Michael Soden, and Markus Scheidgen. A semantic metamodelling framework with simulation and test integration. In *ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing, Bilbao*, 2006.

[124] Jean-Raymond Abrial. Steam-boiler control specification problem. In *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995).*, pages 500–509, London, UK, 1996. Springer-Verlag.

[125] Rajeev Alur and David L. Dill. A theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[126] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[127] Michael Soden. Operational semantics for MOF metamodels: Tutorial on M3Actions. http://www.metamodels.de/docs.html, 2008.

[128] Eclipse Project. *Eclipse Modeling Framework, http://www.eclipse.org/emf.* Last checked: . Januar 2009.

[129] Eclipse Project. *Eclipse Modeling Project (EMP), http://www.eclipse.org/modeling.* Last checked: . Januar 2009.

[130] Michael Soden and Hajo Eichler. Eclipse Proposal: Model Execution Framework, http://www.eclipse.org/proposals/mxf, 2009.

[131] Michael Fisher. Implementing temporal logics: Tools for execution and proof. In *Proceedings of CLIMA VI, LNAI 3900*, pages 129–142. Springer.

[132] Stephan Flake and Wolfgang Mueller. An ASM definition of the dynamic OCL 2.0 semantics. In *UML 2004, volume 3273 of LNCS*, pages 226–240. Springer, 2004.

[133] Hocine EL-Habib Daho and Djilali Benhamamouch. Temporal logic-based modeling and analysis of ASM designs. *Contemporary Engineering Sciences, no. 9-12*, 2:463–478, 2009.

[134] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation - part ii: Single pushout approach and comparison with double pushout approach. In *Handbook of Graph Grammars*, pages 247–312, 1997.

[135] Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electron. Notes Theor. Comput. Sci.*, 253(7):105–120, September 2010.

[136] Peter D. Mosses. Theory and practice of action semantics. In *Proceedings of the 21st International Symposium on Mathematical Foundations of Computer Science*, MFCS '96, pages 37–61, London, UK, UK, 1996. Springer-Verlag.

[137] ikv++ technologies ag. *medini QVT Engine*. http://projects.ikv.de/qvt.

[138] Yuri Gurevich. Asm guide 97. *CSE Technical Report CSE-TR-336-97*, 1997.

[139] Egon Börger, editor. *Modeling with Abstract State Machines: A support for accurate system design and analysis*, B. Rumpe and W. Hesse, editors, Modellierung 2004, volume P-45 of GI-Edition Lecture Notes in Informatics. Springer, 2004.

[140] Egon Börger. The abstract state machines method for high-level system design and analysis, 2003.

[141] Yuri Gurevich and Marc Spielmann. Recursive abstract state machines. *J. Univ. Comput. Sci*, pages 233–246, 1997.

[142] Stephen Brookes. Deconstructing ccs and csp asynchronous communication, fairness, and full abstraction, April 2000.

[143] Andrew Watson. Visual modelling: past, present and future (whitepaper). http://www.uml.org/Visual_Modeling.pdf (15.12.2012).

[144] Domain Technology Committee Technical Architecture by the Platform Technology Committee and Architecture Board. Model driven architecture guide (MDA). a technical perspective. June 2001. ormsc/2001-07-01.

[145] MetaCase. *MetaEdit+. http://www.metacase.com.*

[146] Tanja Mayerhofer, Philip Langer, and Manuel Wimmer. Towards xMOF: Executable dsmls based on fuml. In *Proceedings of the 12th Workshop on Domain-Specific Modeling (DSM'12)*. Online Publication, 2012.

[147] OMG. *Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0*. Object Management Group, feb 2011. formal/2011-02-01.

[148] Guido Wachsmuth. Modelling the operational semantics of domain-specific modelling languages. pages 506–520, 2008.

[149] Eclipse. *Model-to-Model Transformation, http://www.eclipse.org/m2m/*. Last checked: . Januar 2009.

[150] Andy Schürr. PROGRES, a visual language and environment for PROgramming with graph REwriting systems, 1994.

[151] Davide Di Ruscio, Frederic Jouault, Ivan Kurtev, Jean Bevin, and Alfonso Pierantonio. Extending amma for supporting dynamic semantics specifications of dsls. Technical report, Universite Studi dell'Aquila, 2006.

[152] Daniel A. Sadilek and Guido Wachsmuth. Prototyping visual interpreters and debuggers for domain-specific modelling languages. In *ECMDA-FA*, Lecture Notes in Computer Science. Springer, 2008.

[153] Stefan Conrad and Klaus Turowski. Temporal OCL meeting specification demands for business components. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 151–165. 2001.

[154] Sita Ramakrishnan and John Mcgregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.

[155] Paul Ziemann and Martin Gogolla. An OCL extension for formulating temporal constraints. Technical report, Universität Bremen, 2003.

[156] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. On a temporal logic for object-based systems. In *Formal Methods for Open Objectbased Distributed Systems*, pages 305–326. Kluwer Academic Publishers, 2000.

[157] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Towards model checking OCL. In *In Proceedings, ECOOP Workshop on a Precise Semantics for UML*, 2000.

[158] Stephan Flake and Wolfgang Mueller. An OCL extension for real-time constraints. In *Advances in Object Modelling with the OCL, Lecture Notes in Computer Science*, pages 150–171. Springer, 2001.

[159] María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In *FME '02: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 390–409, London, UK, 2002. Springer-Verlag.